# Continual learning for image classification

Anuvabh Dutt

Communauté
UNIVERSITÉ Grenoble Alpes

# THÈSE

Pour obtenir le grade de

**DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES**

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

**Anuvabh Dutt**

Thèse dirigée par **Georges Quénot et Denis Pellerin**

préparée au sein du **Laboratoire d'Informatique de Grenoble et Grenoble Images Parole Signal Automatique Laboratoire** dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

# Continual learning for image classification

## Apprentissage continu pour la classification des images

Thèse soutenue publiquement le **17 décembre 2019**, devant le jury composé de :

**Mme. Jenny Benois-Pineau**
Professeur, Université Bordeaux, Rapporteur

**M. Nicolas Thome**
Professeur, CNAM, Rapporteur

**M. Hervé Le Borgne**
Chercheur, CEA, Examinateur

**M. Masih-Reza Amini**
Professeur, Université Grenoble Alpes, Président

**M. Denis Pellerin**
Professeur, Université Grenoble Alpes, Co-directeur de thèse

**M. Georges Quénot**
Directeur de recherche, CNRS, Directeur de thèse

# Abstract

This thesis deals with deep learning applied to image classification tasks. The primary motivation for this work is to make current deep learning techniques more efficient and to deal with changes in the data and label distribution. We work in the broad framework of continual learning, with the aim to have in the future machine learning models that can continuously improve.

The first contribution involves change in label space of a data set, with the data samples themselves remaining the same. We consider a semantic label hierarchy to which the labels belong. We investigate how we can utilise this hierarchy for obtaining improvements in models which were trained on different levels of this hierarchy.

The second contribution involves continual learning using a generative model. We analyse the usability of samples from a generative model in the case of training good discriminative classifiers. We propose techniques to improve the selection and generation of samples from a generative model. Following this, we observe that continual learning algorithms do undergo some loss in performance when trained on several tasks sequentially. For the third contribution, we analyse the training dynamics in this scenario and compare with training on several tasks simultaneously. We make observations that point to potential difficulties in the learning of models in a continual learning scenario.

Finally for the fourth contribution, we propose a new design template for convolutional networks. This architecture leads to training of smaller models without compromising performance. In addition the design lends itself to easy parallelisation, leading to efficient distributed training.

In conclusion, we looked at two different types of continual learning scenarios and we proposed methods that lead to improvements. Our analysis also points to underlying issues that occur while training in a continual learning scenario. In order to overcome these we provided pointers to changes required in the training scheme of neural networks.

# Resumé

Cette thèse traite de l'apprentissage profond appliqué aux tâches de classification d'images. La principale motivation du travail est de rendre les techniques d'apprentissage profond actuelles plus efficaces et de faire face aux changements dans la distribution des données et les étiquettes. Nous travaillons dans le cadre de l'apprentissage continu, dans le but d'obtenir des modèles d'apprentissage automatique pouvant être améliorés en permanence.

La première contribution implique une modification de l'espace des étiquettes d'un ensemble de données, les échantillons de données restant les mêmes. Nous considérons une hiérarchie d'étiquettes sémantiques. Nous montrons comment il est possible d'utiliser cette hiérarchie pour obtenir une amélioration des modèles formés à différents niveaux de cette hiérarchie.

La deuxième contribution implique un apprentissage continu exploitant un modèle génératif. Nous analysons la possibilité d'utiliser des échantillons issus d'un modèle génératif pour obtenir de bons classifieurs discriminants. Nous proposons ainsi des techniques pour améliorer la sélection et la génération d'échantillons à partir d'un modèle génératif. Enfin, nous observons que les algorithmes d'apprentissage continu subissent certaines pertes de performances lorsqu'ils sont entraînés séquentiellement à plusieurs tâches. Pour la troisième contribution, nous analysons la dynamique de l'apprentissage dans ce scénario et comparons avec l'apprentissage sur plusieurs tâches simultanément. Nous faisons des observations qui indiquent des difficultés potentielles dans l'apprentissage de modèles dans un scénario d'apprentissage continu.

Pour la quatrième contribution, nous proposons un nouveau type d'architectue pour les réseaux convolutifs. Cette architecture permet d'entrainer des modèles plus petits sans perte de performances. De plus, cette architecture se prête facilement à la parallélisation, ce qui permet un apprentissage distribué efficace.

En conclusion, nous avons examiné deux types de scénarios d'apprentissage continu et nous proposons des méthodes qui conduisent à des améliorations. Notre analyse a mis également en évidence des problèmes plus importants, qui laissent penser que nous aurions peut-être besoin de changements dans notre procédure actuelle d'apprentissage de réseau neuronal.

# Contents

# Chapter 1

# Introduction

In this chapter we present the general theme of the thesis and motivation behind the research questions that we ask. We summarise the contributions and present the outline of the rest of the thesis.

## 1.1 Machine learning and neural networks

Machine learning is a sub-field of artificial intelligence (AI) that is concerned with creating systems for solving a task, by providing the system with data related to that task. This paradigm is useful when we know what task to solve but it is hard to elucidate exactly how to solve the task. Consider the task of a cat detection system, which outputs yes or no, depending on the presence of a cat in a photograph. As humans we know the concept of a cat but there is a high chance that we cannot list out all the factors that we used to conclude if a given object is a cat. In such situations we can leverage machine learning to let a model figure out what constitutes the concept of a cat given pictures with and without cats. After learning the model, we can think of the model encoding the concept of a cat. It can also be thought of as a program that is specified by the parameters of this model. In contrast if we wrote a program, for example in python, we would write down all the aspects that we think constitute a cat. In the case of the learned model, these aspects are encoded inside the weights.

Machine learning has greatly matured over the past few years and has had success in various fields such as image classification [54], speech recognition [42], playing Go [95] and beating professionals at poker [5]. Human interaction with machine learning systems is quite ubiquitous now as such systems are used to provide internet search results, language translation, display news feed in social network systems, provide product recommendations on e-commerce sites. Outside of the immediate technological field, machine learning is increasingly being used

in the other domains for tasks such as analysing radiography images, predicting protein structure, and designing molecules.

We have a variety of machine learning models, and out of these neural network models have emerged as the model of choice in a variety of tasks. Architectural and algorithmic advances have made the training of neural networks possible on a diverse range of data. The main advantage of neural networks is that they are able to extract patters from data and learn very good representations. In a lot of cases, such as for image and text, these representations have been found to be much superior to human engineered ones. The ability to learn features from a large variety of data sources have made neural network models a core component of machine learning systems. It seems plausible that as we research and develop more powerful AI systems, neural networks will play a central role, at least for the foreseeable future.

## 1.2 Neural network caveats

Neural networks learn good representations for a variety of tasks but in most cases there is a need for a substantially large training data set. We also require labels for some tasks, particularly when dealing with images. Current research has shown that deep learning models scale well with size, which means that bigger models when given enough data show vast increase in performance. This philosophy of scaling up current techniques has obtained impressive, and somewhat unexpected, results in game-playing [95] and language tasks [14]. Scaling up the training procedure by using larger neural network models and bigger data sets raises computational issues. Training of neural networks in some cases is expensive both in terms of financial cost as well as time. While this may not be a significant obstacle for large organisations, it will surely be a roadblock for the majority of practitioners and researchers. Machine learning and deep learning in particular are starting to be used by people outside of the computer science community and it is not expected that large scale computing resources will be equally available. This is an important motivation towards making neural networks more efficient. Moreover as the data requirements will increase we will also be faced with the question of how to annotate and store this data.

The current paradigm of machine learning systems is to train a model, and then deploy this fixed model. While deployed, this model cannot adapt itself if the data on which it was trained gradually changes. A step towards more powerful systems will be procedures where the system is able to detect this change, acquire data and undergo re-training to adapt its parameters to the current data distribution. This paradigm is usually referred to as incremental learning or continual learning [82]. In this thesis we use the term continual learning because the community

is currently using this term. This is an important area of research if we want to enable a continuously improving and adaptive machine learning model. In our case we will look at continual (incremental) learning from the perspective of change in target space, change in data distribution, and how training models in a continual (incremental) learning scenario differs from standard model training.

There is a general need to make the training of machine learning models efficient, with respect to different parameters such as the size of these networks and the time and data required to train them. Moreover we need to have self adaptive and continuously learning systems that can adapt with changes in the data distribution.

## 1.3 Research questions and contributions

In this thesis we concentrate on supervised learning problems, where we are given a data set of sample and target label pairs. The goal is to learn a model that can associate the correct label when an unseen data sample is presented. We focus on some questions related to some aspects of continual learning in various scenarios and design of more efficient neural network architectures.

**Change in label space of data** We start our discussion with investigating how we can adapt a trained model to different granularity of target labels. The target labels can belong to a semantic hierarchy, for example organised as a tree. One example could be a label which is 'cat' and its associated *child* nodes are various species of cats. From a practical viewpoint, we can initially have a large data set labelled at a coarse scale, such as all species of cats having the same 'cat' label. With time, and more resources, the data set can get annotations at a finer scale and the cats can be more accurately labelled with the correct species labels. In this case, how should we adapt our trained models, and can we take advantage of multiple models trained at different levels of the semantic hierarchy of labels? This is one aspect of continual learning where we have changes in the label space, with the data samples remaining unchanged.

**Change in data distribution** We then consider the more general scenario where a model needs to adapt to a new data distribution, after being trained on a different older data distribution. This is unlike the previous case where only the label space changed. For example, we have a classifier that can identify different categories of cats, it is then presented with a data set of dogs, and is tasked to classify both dogs and cats. This is a continual learning scenario where the model is expected to classify over an increasing number of tasks, *without forgetting* how to perform older tasks. One general approach to achieve this is to store the data from

older tasks, make an expanded data set whenever new tasks and their data arrive, and re-train the model. We investigate in detail the training dynamics of this situation and the effect of the interaction between tasks on the global performance of the model.

**Training without access to old task data** Continuing with the scenario of continual learning, we can see that as data sets get bigger and bigger, it will become impractical to store all of the data. In some situations, such as in embedded devices, it is already quite difficult to store a large amount of data. In place of real data, a generative model can be used, which acts as a compressed version of the original data set. We investigate the current state-of-the-art generative models in the context of using them for training of good classifiers. We discover that the ability to generate realistic images does not necessarily correlate with training of good classifiers.

**Efficient neural network architecture** Finally, our experiments on utilising different levels of a label hierarchy led to insights on a possible design of neural networks for image tasks. We propose a new design template for neural networks that leads to smaller and more efficient models without compromising performance. The design proposed also has advantages for utilising large distributed clusters of machines without excessive communication costs among the machines. As the architecture of the model is quite critical across a wide range of applications, improved architectures will compliment better training algorithms to a great extent, whether it be in a continual learning scenario or a standard one time training setting.

**Contributions of this thesis** We highlighted some issues with neural networks and briefly introduced the research questions that we ask in this thesis. For each question, we have introduced solutions which we hope to serve as stepping stones towards further improvements in the field of machine learning. We summarise the main contributions of this thesis as follows:

- A classifier adaptation algorithm to utilise predictions from a semantic hierarchy of labels. We show how enforcing a constraint on the probabilities output by models at the different hierarchical levels can be used to improve the performance of classifiers.
- We look at training of classifiers from synthetic data as a proxy for real data sets, in a continual learning scenario. We show that current generative models of data have limitations in modelling real world data sets. We propose

a training scheme in order to reduce the gap between real data and synthetic data in terms of the ability to train good classifiers.

- Investigation of the continual learning of a classifier in an image classification task. We empirically show that training classifiers on different tasks sequentially leads to sub-optimal performance. We investigate the reasons behind this. To the best of our knowledge, this perspective of continual learning has not been discussed before in the literature.

- A design for feed-forward convolution architectures that is aimed at reducing the number of parameters and training time, without compromising performance.

**Organisation of this thesis** This thesis is organised as follows:

- Chapter 2 introduces and explains the concepts needed to understand each contribution. We give a general overview of supervised learning, neural networks, and the various types of neural networks considered in this thesis.

- Chapter 3 presents the idea of classifier prediction adjustment using a semantic hierarchy of categories. This is motivated by the question: How do we refine the target categories of a classifier into finer sub categories? This deals with modifying the output probabilities of a classifier so as to be consistent with a given semantic hierarchical relationship among the target categories. This technique can also be thought of as a way to inject domain knowledge into a machine learning system. This work was published in [21, 20].

- Chapter 4 starts with an introduction to the various approaches of continual learning. We then investigate using generative models as replay buffers. Replay buffers can be used to store data from older tasks when a model is being trained on newer tasks. This will need to be done when the data sets get increasingly large and it will not be possible to store all data from all tasks. We use generative adversarial networks to model the training data and explore the gap between training on real data and synthetic samples from such a generative model. This work was published in [11].

- Chapter 4 then continues with an empirical investigation of continual learning using replay buffers. If tasks arrive sequentially how should the training proceed? We take a baseline approach where all data from all tasks are allowed to be stored. Experiments on the sequential training of models are performed and we uncover that this method of training might have difficulty in obtaining the global minimum for all tasks. The article for this work is in preparation for submission to a venue.

- Chapter 5 presents coupled ensembles of neural networks, a new convolutional network design. We will see in Chapter 3 that a notion of parallel computation paths leads to improvement in the performance of a classifier. In this chapter we will systematically investigate this design pattern and demonstrate its advantages over existing architectures, in terms of generalisation error, model compression and ability to parallelise the training. This work was published in [19].

- Chapter 6 summarises the contributions and discusses lines of future research. We discuss the perspectives gained from each contribution and potential future lines of work to extend and improve the contributions.

**Context of this work.** The work done for this thesis was carried out at two laboratories, MRIM team at Laboratoire d'Informatique de Grenoble (LIG) and the AGPIG team of Grenoble Images Parole Signal Automatique Laboratoire (GIPSA-Lab). Both of these laboratories are associated with Université Grenoble Alpes, Grenoble-INP and CNRS. The work was funded by the DeCoRe[1] project from the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01).

[1] https://project.inria.fr/decore/

# Chapter 2

# Background

In this chapter, our aim is to present in a concise manner all background concepts required for understanding the next chapter which comprise the contributions made in this thesis. In addition to this chapter, each specific chapter will include specific details pertaining to its content.

## 2.1 Machine learning

Machine learning can be broadly defined as discovering patterns in data and then using these learned patterns to make useful predictions when given new data that has not been seen before. Machine learning has two main elements: the model and the data. We want to learn the parameters of the model given some data. Note that some models may not have parameters in the strict sense but rather the model itself contains rules to operate on the data ($kNN$ classifier is an example and we will discuss it shortly).

One can also think of machine learning as discovering a computer program. The reason we said that is like a program is that at the the end the machine learning model will be used to perform a task, and in order to perform this task, the steps will be learnt and encoded in the parameters of the model. This is similar to the program one writes when asked to solve a task, for example given a a word check if it is a palindrome or not. In this case we know that a string is a palindrome is if the sequence of characters is unchanged when the word is reversed. Checking for a palindrome is a task where we know the *exact* rules and can express these rules in the form of a computer program. If we move onto a task such as detecting a cat in in image, things get a bit more complicated. We still know if a cat is present in an image or not but we are not able to express our internal decision making as concretely as in the case of the palindrome checking task.

Just as in the case of the cat-not-a-cat task, the real world presents several

complex tasks which we can perform but are not able to articulate the exact and general way to specify a solution to these tasks. Our mental model of a cat is quite powerful enough to generalise to most unseen cats. However there is really a lot of variation among cats and this fact is compounded further by the fact the images can be captures by different cameras in different lighting conditions. All this mounts to a truly monumental variation in the ways we can capture the image of cats. We can instead devise algorithms that can learn from examples. In the most common case of supervised machine learning we construct a model by given it data samples which we label. Our learning algorithm then discovers patterns to solve the task. The final model is analogous to our computer program. Machine learning has been developed to a great degree and this philosophy of constructing models has made many difficult tasks tractable and indeed solved by computer to a super human level in some cases.

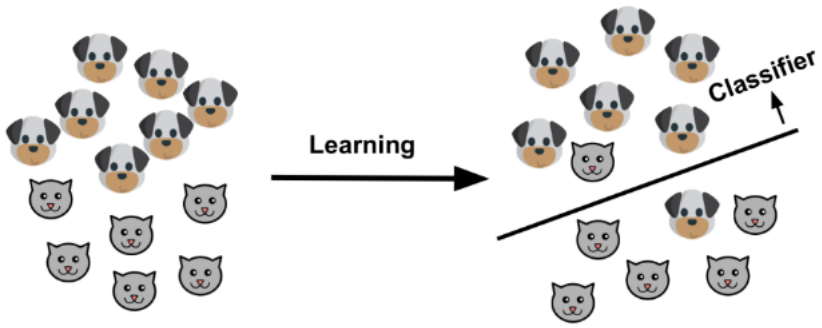
### 2.1.1 Categories of machine learning

Machine learning models and training algorithms can be categorised in a myriad ways. Here we review the most common characterisation which is in terms of the nature of data presented to the model.

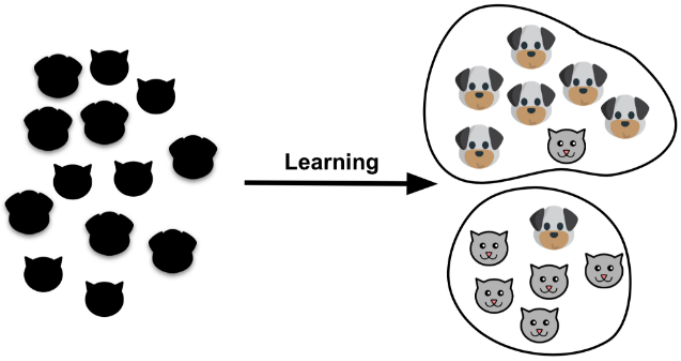
**Supervised learning** This is probably the most practised form of machine learning. Our dataset $\{(\mathbf{x}_i, \mathbf{y}_i); i = 1..n\}$, is a set of tuples. Each tuple comprises $\mathbf{x}_i$, a data sample, and $\mathbf{y}_i$ its corresponding label. The model learns a mapping $x \in R^D \rightarrow y \in R^{D'}$. In slightly more formal terms, if your data set is sampled from a distribution $\mathcal{D}$, our model is defined by parameters $\theta$, we want to estimate the conditional probability $p(y|x, \theta)$.

**Unsupervised learning** The data set is of the form $D = \{(\mathbf{x}_i); i = 1..n\}$. No labels are known and the the model has to group given data samples and discover the categories. This problem is quite difficult to evaluate because there can be multiple different groupings. In order for the model to discover useful patterns these models usually are task specific. Formally, in unsupervised learning we want to estimate $p(x|\theta)$, and this is often referred to as density estimation. See Figure 2.1 for a visual depiction of supervised and unsupervised learning.

**Reinforcement learning** In this case instead of having tuples of data and labels, we have an environment which we can observe, take actions, and get rewards from the environment. The model learns to take actions conditioned on the environment state and previous actions so as to maximise its rewards. This is a bit different from finding patterns and assigning categories to new data points. It is




(a) Supervised learning




(b) Unsupervised learning

Figure 2.1 – Supervised and unsupervised learning toy example.[1]

a dynamic scenario where the model predictions influence the future data points and actions to be taken.

## 2.2 Supervised learning

This thesis mainly involves algorithms and models for supervised learning. In this section we present the fundamentals of supervised learning. We go over the notion of loss functions and the principle on which we optimise the model parameters to minimise a loss function. We then present linear regression which will serve as a stepping stone towards building neural networks.

**Empirical risk minimisation (ERM)** In supervised learning our objective is to learn a model or a function from a set of labelled examples of the form $S = \{(x_i, y_i)\}_{i=1}^{N}$. The examples belong to some unknown distribution $\mathcal{D}$ and have been sampled independently of each other. The samples are said to *independently and*

[1]Figure taken from https://hal.archives-ouvertes.fr/tel-01881069

*identically distributed (iid)*. We want to learn a classifier that can predict the labels of unknown samples with good performance, as measured by some criteria. For that we define a hypothesis space of classifiers, $\mathcal{H}$, such that $\forall h \in \mathcal{H}, h : \mathcal{X} \to \mathcal{Y}$. We want to choose from the space of $\mathcal{H}$, using the samples in $S$, the classifier $h_S$ that will predict the labels of unknown examples sampled from $\mathcal{D}$. In order to evaluate the quality of a classifier $h_S$ we need a criterion, which can be defined through the notion of 'true risk':

$$\mathbf{R}_D(h) = \mathop{\mathbb{E}}_{(x,y\sim\mathcal{D})} \mathbb{1}_{[h(x)\neq y]} \tag{2.1}$$

$\mathbb{1}_{[p]}$ is 1 when the predicate $p$ is true else 0. This is an example of a *loss function*. Loss functions play a critical role not only in evaluating a classifier but also in training of the parameters of the classifier. The goal is to find the classifier with minimum true risk. However the true risk cannot be computed since the distribution $\mathcal{D}$ is unknown and we only have access to the samples in $S$. Hence we compute the 'empirical risk':

$$\mathbf{R}_S(h) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{[h(x_i)\neq y_i]} \tag{2.2}$$

In practice we choose a classifier which minimised the empirical risk but does not deviate too much from the true risk. The deviation can happen because we only have a limited number of training samples and the classifier may *overfit* to these samples. One way to prevent this is to choose *simple* classifiers. We obtain simple classifiers through *regularisation*, either in the form of simple models or restrictions places on more complex models.

**Maximum likelihood estimation** One of the methods used to find the parameters of a statistical model is maximum likelihood estimation (MLE). Following this principle we can derive the loss functions that are used to train our models and obtain the parameters. Formally this is expressed as:

$$\hat{\theta} = \arg\max_{\theta} P(\mathcal{D}|\theta) \tag{2.3}$$

Utilising the *iid* assumption, we can then write the likelihood estimation as:

$$\hat{\theta} = \arg\max_{\theta} \prod_{i} P(y_i|x_i, \theta) \tag{2.4}$$

Instead of working with the product form, we can instead take the log of the expression. Since log is a monotonic function maximising the log also maximised

the original expression. Further instead of maximising the log-likelihood, we can minimise the negative log-likelihood. These transformations make the computation easier to handle with optimisation software package. We can then write the negative log-likelihood (NLL) as:

$$l(\theta) = -\sum_i \log P(y_i|x_i, \theta) \tag{2.5}$$

Equation 2.5 provides the basis for computing the loss functions for various models. If we substitute the expression for $P$ in Equation 2.5 we can derive the specific loss function that needs to be minimised. We will see an example of this next.

**Linear regression** Linear regression is a widely used machine learning model which is simple to understand and provides the foundations for concepts such as neural networks. As the name implies, the model assumes that for a given input the corresponding output is a linear function of the input. We can write this as:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + \epsilon = \sum_j w_j x_j + \epsilon \tag{2.6}$$

$\mathbf{w}$ is the parameter vector with $d$ dimensions. $\mathbf{x}$ is the input with the same number of dimensions $d$. We learn the parameter vector $\mathbf{w}$, and each component $w_i$ acts as weights to each component of the input $\mathbf{x}$. $\epsilon$ is the error between output of the model and true value. We can model non-linear relationships between the input and output, if $\mathbf{x}$ is replaced with a non-linear function of the input $\phi(\mathbf{x})$. A reasonable assumption to make is that the output $y$ follows a normal distribution. This makes the output the conditional probability of the output $y$, given the input and the model parameters. We can re-write the model as:

$$p(y|\mathbf{x}, \mathbf{w}) = \mathcal{N}(y|\mathbf{w}^T\phi(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{2.7}$$

We can now use the MLE principle to find the parameters of out model. If we use this expression in the MLE expression of Equation 2.5 with the additional assumption that noise is constant ($\sigma^2(\mathbf{x}) = \sigma^2$), we find that the log-likelihood can be written as:

$$l(\mathbf{w}) = -\sum_{i=1}^{N} \log \left[ \left( \frac{1}{2\pi\sigma^2} \right)^{\frac{1}{2}} \exp\left( -\frac{1}{2\sigma^2}(\mathbf{y}_i - \mathbf{w}^T\mathbf{x}_i)^2 \right) \right] \tag{2.8}$$

$$l(\mathbf{w}) = \frac{1}{2\sigma^2} RSS(\mathbf{w}) + \frac{N}{2}\log(2\pi\sigma^2) \tag{2.9}$$

RSS is the residual sum of squares and the average RSS is commonly referred to as the mean squared error (MSE). This is a loss function that is quite commonly used in machine learning and here we see that it follows from applying the MLE principle.

$$RSS(\mathbf{w}) = \sum_{i}^{N} (y_i - \mathbf{w}^T \mathbf{x})^2 \tag{2.10}$$

From the above equations we see that the NLL is minimised when the $RSS$ is minimised. This happens as the predictions of the model ($\mathbf{w}^T\mathbf{x}$) get closer to the true output ($y$). This loss function is differentiable and hence can be used with a gradient based learning algorithm such as stochastic gradient descent to iteratively find the parameters of the model [2].

**MAP estimation and regularisation** The MLE estimate of the parameters may result in a situation called *overfitting* which means that the parameters take on values that are are a perfect fit of the training data. This is because our optimisation procedure is geared towards minimising the loss on the training set and we have no restrictions on the values that the parameters can take. Essentially the parameters 'memorise' the training data. This is similar to our discussion on simple and complex hypothesis space of classifiers from the ERM principle. In most cases if the parameter overfits to the training data it will not generalise well to unseen test data.

In order to avoid this we place a restriction on the parameter $\mathbf{w}$. We can encourage the parameter values to be small by enforcing that the weights are sampled from a zero-mean gaussian prior:

$$p(w) = \prod_{j} \mathcal{N}(w_j | 0, \tau^2) \tag{2.11}$$

If we apply the same principle as before we will finally obtain the loss function for linear regression as:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i}^{N} (y_i - \mathbf{w}^T \mathbf{x})^2 + \lambda ||\mathbf{w}||_2^2 \tag{2.12}$$

$\lambda$ controls the strength of the regularisation. This particular regularisation is called $l_2$ regularisation or *weight decay*. Larger weights lead to an higher cost, which means that the optimisation procedure will encourage the weight values to be small. This leads to less complex models which in most cases generalise better. Another way to look at it is regularisation prevents machine learning models from modelling the noise in training data.

[2] A closed form solution exists for linear regression which can be derived through the normal equations and the corresponding ordinary least squares solution.

**Logistic regression** Linear regression is used to obtain real valued outputs given an input. We can use a similar model for classification which we refer to as logistic regression. This name is due to its similarity with linear regression. We make two different assumptions: the output is sampled from a Bernoulli distribution ($Ber$) instead of a Gaussian distribution, and the output space is restricted to $\{0, 1\}$. The restriction is implemented by passing the output through a *sigmoid* function, which squashes the input between 0 and 1, which helps in interpreting this as a probability. Finally we can write the model as:

$$p(y|\mathbf{x}, \mathbf{w}) = Ber(y|sigm(\mathbf{w}^T\mathbf{x}) \tag{2.13}$$

As before, we can apply the MLE principle to estimate the parameters of our model. We can write the negative log-likelihood as:

$$NLL(\mathbf{w}) = -\sum_{i=1}^{N} \log[\mu_i^{\mathbb{I}(y_i=1)} \times (1-\mu_i)^{\mathbb{I}(y_i=0)}] \tag{2.14}$$

$$= -\sum_{i=1}^{N} [y_i \log \mu_i \times (1-y_i) \log(1-\mu_i)] \tag{2.15}$$

The classification is done by thresholding the output at some value (for example 0.5), which forms the decision rule of assigning a category to each unseen test sample.

## 2.3 Neural networks

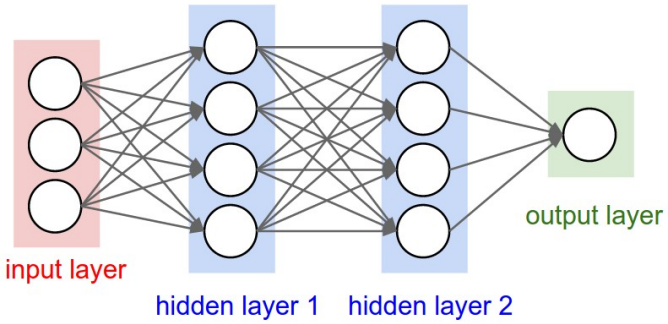



Figure 2.2 – Neural network with 2 hidden layers. [3]

[3]Figure taken from https://cs231n.github.io/neural-networks-1/

We defined linear regression as a model of the form: $f(x) = y = \mathbf{w}^T\mathbf{x}$. Neural networks are compositions of such functions. We can write this as:

$$y = f(g(...h(x))) \tag{2.16}$$

Each of $f, g, h$ can be written as:

$$f(x) = \sigma(\mathbf{w}_f^T\mathbf{x}) \tag{2.17}$$

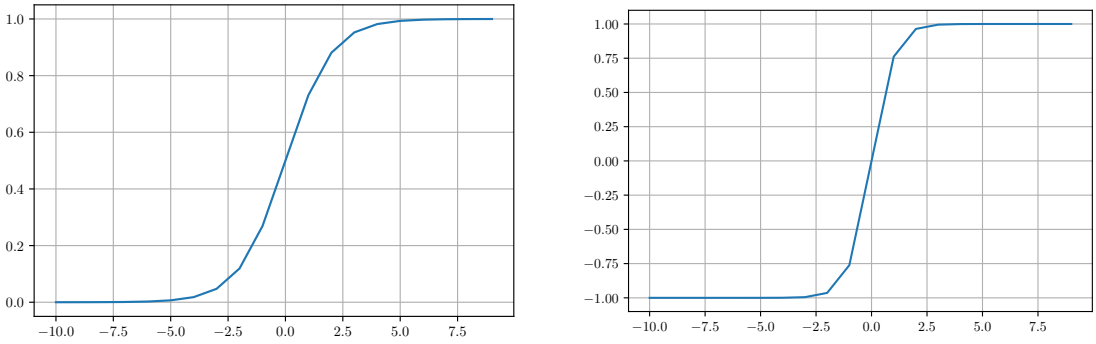
Each function has its own weight vector $\mathbf{w}$. The input is transformed sequentially by each function. The non-linear function $\sigma$, is called the *activation function*. This non-linearity makes it possible to learn a non-linear mapping between the input and output. If the activation function was linear, we can transform each $\mathbf{w}$ into a single equivalent parameter $\mathbf{w}'$. Figure 2.2 shows a visual depiction of a 2 layer neural network

A neural network essentially learns a non-linear function between the input and output. At each step the current input is linearly transformed by the parameters associated with the function and then passed through activation function. The parameters of each of these functions are learned jointly using some learning rule. This general structure makes it suitable for a wide range of tasks. Recent advances in the specific parameterisation of $\mathbf{w}$, training algorithms, regularisation strategies, and large training data sets have made neural networks the model of choice in a wide variety of tasks.

**Depth and width of neural networks** Equation 2.16 shows a composition of functions. From an implementation point of view this can be thought of as the input going through several computation steps sequentially. The number of these computation steps is referred to as the number of *layers* or the *depth* of a neural network. Modern neural networks can have 100s of layers, and hence the term deep learning.

Each layer of the neural network transforms its input by multiplying the input with a weight matrix. For example if the input has $d$ dimensions, the required output has $d'$ dimensions, the weight matrix will have a shape of $d \times d'$. Note that this is in contrast to linear regression where the weight is always a vector of shape $d \times 1$. The *width* of the network at each layer is then equal to $d'$. This is often called as the number of units in each layer.

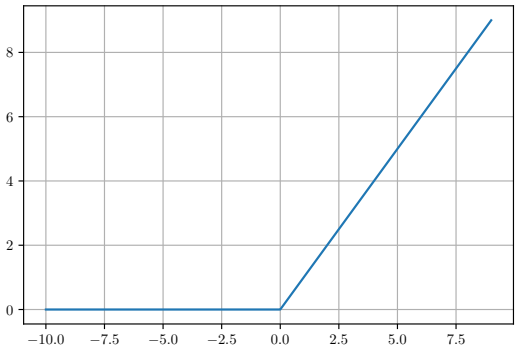
**Activation functions** Activation functions are critical to the functioning of neural networks, especially when the number of layers are large. The weights of each layer are jointly. We will see later that this is done by computing the gradient of the global loss function with respect to each weight vector. In order to



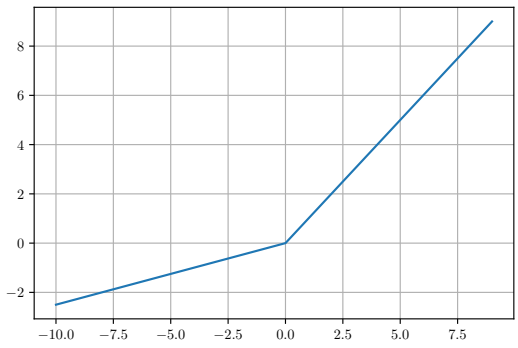

(a) Sigmoid




(b) TanH




(c) ReLU




(d) PReLU, $\alpha = 0.25$

Figure 2.3 – Activation function plots.

compute the gradient efficiently we need the gradient to flow through each layer. The value of the gradient depends on how we transform the input to the layer. Additionally in order to continue training with a sufficient gradient it would be better to have activation functions that are non-saturating. Some commonly used activation functions are:

- **Sigmoid.** The sigmoid function squashes the input to be between 0 and 1. It is defined as:

$$sigm(x) = \frac{1}{1 + \exp(-x)} \tag{2.18}$$

  One issue with the sigmoid function is that inputs with high or low values saturate, that is they are close to 1 and 0 respectively. If we look at the curve of the sigmoid function (Figure 2.3), this means that the gradient is close to 0. This is an issue in gradient based learning algorithms since a gradient of 0 means there is not much training signal.

- **Hyperbolic tangent**. It is similar to the sigmoid but squashes the input

to $\{-1, +1\}$. We still have the same issue with saturation but the range of inputs that do not saturate are higher.

- **ReLU.** This is a linear rectifier, any input that is below 0 is cut-off and all inputs above 0 are unchanged. It is defined as:

$$ReLU(x) = max(0, x) \tag{2.19}$$

The gradient for all inputs greater than 0 is 1. This helps to preserve gradient flow through our network and this activation function is critical in training deep networks. However the gradient is 0 for all units that produce negative outputs. This means that the weights corresponding to these units do not receive any gradient and hence do not learn. To avoid such situations it should be checked that the weight values are initialised properly.

- **Parametric ReLU.** The problem of 0 gradient for negative input values can be tackled by replacing the 0 with a small negative value. We can also make this small value a parameter and learn it with the rest of the parameters of the model. It is defined as:

$$PReLU(x) = \mathbb{1}_{x<0}\ \alpha x + \mathbb{1}_{x\geq 0}\ x \tag{2.20}$$

These are some of the commonly used activation functions. ReLU is the most used function. However the application and architecture of the neural network often dictates which function to use, as for example in LSTM (a type of recurrent neural network), sigmoid and tanh functions are used as a ReLU would make learning unstable and maybe even impossible. There exist other activation functions such as *maxout*, gaussian linear unit, etc.

**Weight initialisation** It is important to initialise the weights of each layer in a manner which prevents both high and low gradient magnitude early in the training. A good initialisation leads to faster training convergence and even good generalisation. One idea would be initialise all weights with 0. The problem with this is that each unit computes the same output value for the input, which in turn means that the gradient is the same for all weights. Hence the weights will learn in the same manner and not lead to useful results. In order to avoid this, weights could be initialised with small values close to 0. Again, too small weight values will lead to small gradients (as the gradient is proportional to the weight value), which hinder training and slow down convergence.

Weight initialisation without accounting for the number of inputs and outputs and each layer leads to a blow-up of the output distribution at the start of training. We can normalise this by scaling the weight vector by the square root of the number

of inputs, which is called the *fan-in*. `xavier` initialisation [26] which is widely used recommends sampling the weights from a normal distribution with zero-mean and variance $2/(fan\_in + fan\_out)$. This derivation is for a linear case and does not take into account the ReLU non-linearity. The recommended initialisation for deep networks with ReLU is the `kaiming` initialisation [36] which samples weights from a normal distribution with zero-mean and variance of $2/fan\_in$.

Each layer can have a *bias* parameter. Empirically it has been seen to fine to initialise the bias values to 0. If using ReLU it is advisable to use small initial values to prevent inactive units at the start of training. There also exists data driven approaches to weight initialisation, for example layer-sequential unit-variance initialisation [71] which takes into account the data set and all layers and non-linearities in the network. However these are a bit expensive to compute and as neural network architectures have converged to a stable template, these category of techniques are not that frequently used.

**Regularisation** Regularisation typically increases training error with the objective of reducing generalisation error. It acts as a mechanism for avoiding fitting model parameters to noise in the training data. Typically deep learning models have quite high number of parameters and hence proper regularisation is quite important. In the following, we discuss some common regularisation strategies used in the context of deep neural networks:

- We have seen in §2.2 the theoretical view of of $L_2$ regularisation, and it has been used for training neural networks for a long time [55]. We can start our discussion with another view of $L_2$ weight penalty, to make clear how it encourages weight values to remain small. $L_2$ regularisation on the weights prevents weights from taking on large values, which is one way to keep models simple. The cost function with $L_2$ norm penalty is $J(w) + (1/2)\lambda||w||_2^2$. The gradient for the cost with respect to the weights is: $\nabla_w J(w) + \lambda w$. One step of gradient descent can then be written as:

$$w_t = w_{t-1} - \eta_t(\nabla_w J(w_{t-1}) + \lambda w) \tag{2.21}$$
$$w_t = (w_{t-1} - \eta_t \lambda w_{t-1}) - \eta_t \nabla_w J(w_{t-1}) \tag{2.22}$$
$$w_t = (1 - \eta_t \lambda) w_{t-1} - \eta_t \nabla_w J(w_{t-1}) \tag{2.23}$$

  We see that at every weight update step, the weights are reduced by a constant multiplicative factor, $1 - \eta_t\lambda$.

- **Data set augmentation** can also be thought of as a form of regularisation. It is almost ubiquitous in training of neural networks for image tasks. For

example, during classification, data augmentation can comprise random flips, translation, colour space augmentations, etc. These help to increase the effective number of training samples and account of different test examples that we may expect.

- **Early stopping** [73] is a widely used, simple regularisation strategy. It is simple to implement, during training we keep track of the loss on a validation set, and we stop training when the loss increases for a pre-defined number of optimisation steps. Early stopping requires little overhead besides keeping track of the parameter values at the steps before the loss started increasing. Since we are training for a restricted number of epochs, it prevents gradient descent from exploring all of the parameter space and restricts it to the parameter space close to the starting point.

- **Addition of noise** to the training procedure has also been used as a form of regularisation. Noise with a small norm added to the input is similar to data augmentation. In de-noising autoencoders, this has been used to train an model to *de-noise* input images. Another way is to add small amount of noise to the weight at each update step. Addition of gradient noise has also been investigated and found useful to improve generalisation of some models such as Neural Turing Machines and Neural GPUs. The mini-batch training procedure is also a source of noise, since the gradient that is computed is an approximation of the exact gradient over the entire training data set, and hence inherently noisy. It has been shown that smaller batch sizes, which are inherently more noisy, generalise better compared to larger batch sizes. One hypothesis is that training with noise leads to solutions with *wider minima* [43].

- **Dropout** [98] is a way to train an exponential ensemble of neural networks with shared weights, and in practice serves as a regulariser. Dropout out works by stochastically dropping some connections at each training step. The probability of dropping a connection is a hyperparameter, called the *drop probability* (usually 0.5 for the hidden units). The intuition is that since any connection can be dropped each unit must learn to produce an good output without relying on all incoming connections. This also implies that connections are encouraged to learn features that are useful in a variety of contexts. More formally, this has similarities to bagging methods, where outputs of an ensemble are used to reduce the variance in predictions from individual models. During inference of models trained with dropout, the weights are scaled by the drop probability in order to have a normalised output distribution.

**Backpropagation** Backpropagation [86] is the algorithm used to efficiently compute the gradient of the loss with respect to each parameter in the neural network. It is a form of dynamic programming, the gradient at each layer of the network depends on the local gradient between the consecutive layers and the gradient of the following layer.

**Optimisers** Neural networks are trained with some variety of gradient descent. Each optimiser uses the gradient that is computed by backpropagation to determine the weight update. The common algorithms include stochastic gradient descent (SGD) and Adam. SGD is probably the most widely used when training convolutional networks on image tasks, whereas Adam is almost exclusively used for training generative adversarial networks (These family of models are described in detail in the next sections).

SGD with momentum is widely used for training a variety of different neural network models. Using momentum weight updates smoothens the optimisation trajectory and prevents bouncing around in the loss surface especially early in training. One weight update step is defined as:

$$v_t = \mu v_{t-1} - \eta_t \nabla_w J(w_{t-1}) \tag{2.24}$$

$$w_t = w_{t-1} + v_t \tag{2.25}$$

During training of deep neural networks, the learning rate, $\eta_t$ needs to be adjusted. This is called the learning rate decay schedule. One popular schedule is a step decay schedule for convolutional networks. SGD with such a schedule obtains very good generalisation across a wide range of tasks and is quite hard to beat. Some works have focused on choosing the correct learning rate and decay schedules to greatly speed up training convergence [66].

Adam [51] is another widely used optimiser. The weight update rule of Adam takes into account the history of updates and gradient for each parameter individually. This means that the effective learning rate is set per-parameter, as compared to a global learning rate in the case of SGD. A global step size can still be set for the algorithm and empirically it has been seen that a few choices of this parameter work across several tasks. Hence there is not much need to search for a good learning rate and a proper learning rate schedule for each specific task. This properties make it suitable for training generative adversarial networks, which are relatively less stable than training of other models.

**Training procedure** Now that we have described the various building blocks of a neural network model and the associated training elements, we can outline

how the weights of a neural network are trained. This description is for supervised training but most concepts still apply to other training paradigms.

The training starts by sampling a batch of data, $\mathcal{B}$, from the data set. The effect of the size of $\mathcal{B}$ has been studied with several works recommending that smaller sizes lead to better generalisation. However, it leads to increased computation time, as current hardware is optimised for dense computations. A common batch size is 64.

$\mathcal{B}$ is input to the model, the computations are done at each layer and finally we obtain the output, $\hat{y}$. The loss function is applied to the output and the ground truth labels, $y$, to obtain the loss $\mathcal{L}(y, \hat{y})$. This is called the *forward pass* through the network.

The loss is used to calculate the gradient for each parameter. This is done efficiently through the backpropagation algorithm. In most software packages this is implemented using reverse mode automatic differentiation. This step is called the *backward pass* through the network. After the backward pass, each parameter has the error gradient with respect to it. This gradient is used by the chosen optimiser to update the weights.

The above steps continue until we have gone over the entire data set, at which step we say that one *epoch* has completed. The learning rate for the optimiser is adjusted as training proceeds. In order for the training to converge several epochs are computed. One criteria to determine convergence is when the loss on a held out validation data set plateaus.

**Types of neural networks** Neural networks can be categorised depending on the nature of computations performed on the inputs and the specific parameterisation of the weights at each layer. The most general kind is a *feed-forward network* in which the input is applied at the first layer, and then sequential operations are performed by transform the input consecutively with different weights. It is called feed-forward, because there are no connections that *go back* among the layers, each layer computes an output and provides it as input to layers that come after it.

If we share the weight parameter at each layer, we obtain a *recurrent neural network*. Usually this type of neural network operates on sequential data, such as text or time-series. Each step of the sequence is given as input to the network which produces an output. This output is then combined with the next step of the sequence and given back as input to the network. It is called recurrent because we can think of this architecture which applies the same weights repeatedly across the input sequence.

These two types comprise the broad family of architecture. For both feed-forward and recurrent, we have various special types of networks. In this work we focus mainly on a specific type of feed-forward models, called convolutional

networks. We describe this in detail in the next section.

## 2.4 Convolutional networks

Convolutional networks (CNN) [58] are a type of feed-forward neural network. These networks are used most for image based tasks, though recently they are starting to be used in sequence based tasks such as machine translation. Convolutional networks use the *convolution* operation to compute the output, given an input. This is in contrast to the matrix multiplication between the weight and input that we have seen till now. In images, this operation takes advantage that patterns can appear in local neighbourhoods in a given point in an image. To exploit this fact, convolution operation operates on such local neighbourhoods, instead of operating on the entire spatial dimension directly. This leads to vast reduction of parameters, and consequently in the size of the model. We first describe the convolution and pooling operations, and then describe how a convolutional network is constructed.

**Convolution operation** Mathematically, given an *input* $x$, a *kernel* $w$, the *output* $s$ as a result of convolution of $x$ with $w$ is:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{2.26}$$

The output is also referred to as the *feature map*. We can think of this as applying a weighted operating around a neighbourhood of the point $t$. The convolution can be extended across multiple axes instead of the single axis shown in this formula. In practice we work with discrete points, such as pixels in images. We can then replace the integral with summation. The resulting convolution operation can then be written as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.27}$$

Convolution is a commutative operation, which means that we can equivalently flip the kernel with respect to the input. This leads to a somewhat equivalent implementation which is usually found in most software packages, and called *cross-correlation*.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.28}$$

In the context of CNNs, each of the kernels are randomly initialised and then learnt from data. In practice after learning the kernels have been see to recognise intuitive features in the lower layers and more abstract higher dimensional

features in the deeper layers of the network. This makes sense, since the layered computation in neural networks implies that we expect a hierarchical composition of features, from simple to more complex.

**Properties of convolution** The use of the convolution operation introduces certain desirable properties in a neural network. Convolution introduces *sparse* interaction between the input and the output because the kernel is usually much smaller than the input. This means that every output point is the result of an operation applied to only a small input region. For instance, if the kernel has $k$ parameters, and the number of output units is $n$, the parameters needed are $\mathcal{O}(k \times n)$. This does not prevent the modelling of patterns across different spatial regions outside the neighbourhood of the kernel, since the network has several layers, and deeper layers will indirectly induce interaction among different spatial regions.

CNNs also require that the same kernel be applied the entire input. This is achieved by having the weight values defining the kernel be the same across the weight matrix. In contrast, without weights being the same, a different weight would be applied at each position of the input. To learn different features, we can define several kernels. Since each kernel is applied to the entire input, CNNs are feature *equivariant* to translation, by design. After the weights have been learnt, the kernel will be able to detect its corresponding feature no matter where in the input it appears.

By design convolution operation is agnostic to the input size. However the output size depends on the convolution operation and if it is input to some other function which is sensitive to input size, it will lead to computational issues. In order to avoid these, the input can be *padded* to make it have the correct dimensions (for example adding pixels with 0 value at the borders of an image).

**Pooling** A pooling operation is used to generated a summary statistic, generally after feature maps are obtained from a convolution operation. Pooling is way to encode feature invariance in the network. Feature invariance means that the network is able to detect if a feature exists or not but is not particularly concerned with where exactly the feature exists. It common to have a *max pool* operation after a few convolutional layers. The convolutional layers detect features, and the pooling layer aggregates these features. Pooling also reduces the spatial dimension, which helps increase in computational load.

Two common pooling operations are *max pool* and *global average pool*. Max pool replaces the input of a rectangular area with the maximum of the input. A common operation is to do max pool over $2 \times 2$ input regions. In convolution

networks, max pool is present after a every few convolution layers. Global average pool operation is usually done just before the final classification layer of the network. It replaces the input spatial region of each feature map with the average value. This helps to summarise the feature maps and reduce the dimensionality over which the final classification is to be done.

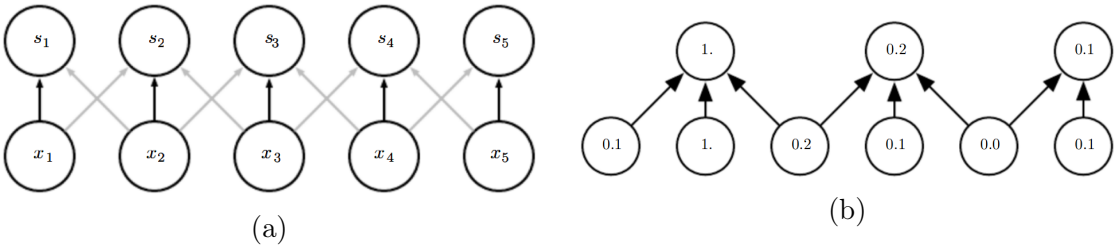



Figure 2.4 – Illustration of convolution and pooling operations. Figure taken from [28].

**Skip connections and batch normalisation** The two other building blocks of modern CNNs are *skip-connections* [35, 99] and *batch normalisation* [48]. Skip-connections are formed when a layer takes the input from multiple previous layers rather than just the immediate previous layer. This can have several implementations [35, 45, 99]. It has been shown to be one of the components needed for training very deep networks, as without skip-connections deeper networks can have worse performance [35]. One reason is that introducing skip-connections, particularly identity skip-connections helps in the gradient flow from the loss function back to the early layers [37]. Figure 2.5 shows an identity skip connection over two layers.

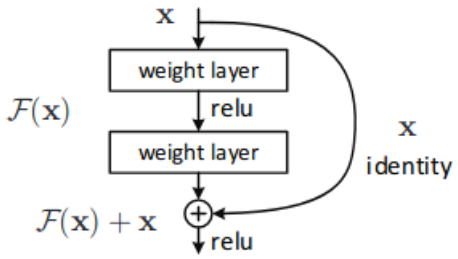



Figure 2.5 – Visual depiction of skip-connections in a deep neural network. Figure taken from [35].

Batch normalisation (`batch_norm`) normalises each component of the output tensor along the batch dimension. For example in the case of convolution layers,

if the output has $f$ feature maps, and the batch size is $b$, each feature map will be normalised by subtracting the mean and dividing by the standard deviation, computed on the batch. The result of this operation is then scaled and translated by two parameters that are learnt from data. If the input to a batch normalisation layers is has $d$-dimensions, $x = (x^1, x^2, ...x^d)$, then the normalisation along each dimension can be written as:

$$\hat{x}^k = \frac{x^k - \mathbb{E}[x^k]}{\sqrt{Var[x^k]}} \tag{2.29}$$

This normalisation would change the output and restrict what each layer can represent. In order to let the network learn parameters that enable it to remain unconstrained so that the correct representation can be learnt, two learnable parameters, $\gamma, \beta$, are introduced which scale and translate the normalised values:

$$y^k = \hat{x}^k \gamma^k + \beta^k \tag{2.30}$$

Batch normalisation has been empirically shown to speed up training, enable training with higher learning rates, and is less sensitive to weight initialisation. The exact mechanics of how batch normalisation helps training is not understood completely yet and is being studied [92].

The normalising statistics are calculated over the mini-batch and hence the number of samples in each mini-batch should be sufficiently high such that the statistics do not have very high variance. During training, each batch norm layer maintains a moving average of the mean and variance of the mini-batch statistics. During testing, each batch of test samples are normalised with these stored moving averages rather than the mini-batch statistics. This ensures a deterministic output for every input, rather than coupling each output to other input samples in the batch. When working with smaller batch sizes we can look at normalisation techniques such as "batch renormalisation" [47] and group normalisation [106]. These are more suited when we are constrained to smaller batch sizes, for example in tasks such as segmentation where larger batch sizes are often not possible due to memory constraints. Some other normalisation schemes that have been proposed for training neural networks are layer normalisation [3] and weight normalisation [89] .

**CNN architecture** We can outline a generic CNN architecture using the building blocks described. We start with a 3 layer units which comprises: convolution, batch norm, non-linearity. These 3 layers are repeated several times. Usually the convolution includes an appropriate stride and kernel size so the the spatial dimensions of the input do not change. After a few of these layers, we have a pooling layer which aggregates the computed features and reduces the spatial dimension.

This entire sequence of layers is then repeated two or three times, depending on the input resolution. Finally we have a global average pooling layer which summarises the final feature maps. This is then fed to a fully connected layer which outputs score vector for the target categories. Further details are given in § 5.1.

## 2.5 Generative adversarial networks

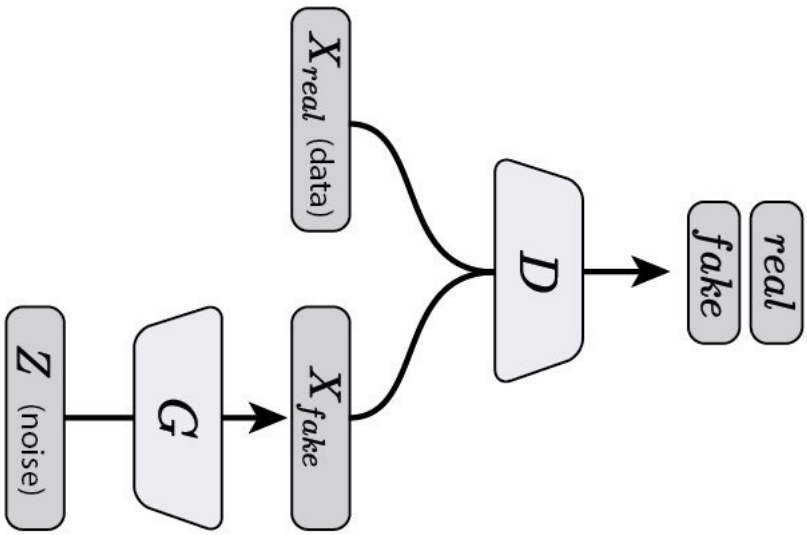



Figure 2.6 – Schematic of a GAN model. [4]

Generative adversarial networks (GAN) [29] are a category of generative models, which try to model the data set distribution $P(x)$ instead of the conditional distribution $P(y|x)$, such as in the case of discriminative classifiers. Here $x, y$ are the sample and corresponding label respectively. More specifically GAN networks are a category of differential generative networks, where the generator is implemented as a neural network and the parameters of the network are trained through some variant of gradient descent.

A GAN network has two component networks, which are trained by making the two components compete against each other (see Figure 2.6). There is a *generator* network which maps a noise vector to a data sample $\hat{\mathbf{x}} = G(\mathbf{z}; \theta^G)$. The second component is a *discriminator* network that tries to distinguish between real training samples and generated samples. It outputs the probability of a sample belonging to the real training data distribution as $D(\mathbf{x}; \theta^D)$). Both the generator and discriminator are implemented as neural networks. As training proceeds, the

[4] Figure taken from https://twitter.com/ch402/status/793535193835417601/photo/1

discriminator learns to distinguish between real and fake samples. In parallel the generator tries to produce samples that are harder for the discriminator to distinguish, and one way to do that is produce samples that resemble the real training data. Ideally at convergence, the discriminator should output a probability of 1/2 for all samples. We can write the objective function for training GANs as (taken from [29]):

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p(z)}[\log(1 - D(G(z)))] \tag{2.31}$$

In practice the training proceeds as follows: we train $D$ for a few iterations, $k$, to distinguish between real and fake samples. Then one weight update step is taken for $G$. This iterative manner of training is less computationally expensive than first finding the optimal parameter for $D$, which would also likely result in overfitting. The iterative procedure lets the $D$ parameters be optimal for the current setting, and also provides meaningful gradient signal to train the $G$ parameters. Notice the $G$ is trained through the loss on output of $D$, which means that $D$ should provide enough gradient signal such that $G$ parameters can be updated. On the other hand if the difference between $G$ and $D$ is too much, $D$ would just reject all samples from $G$ and we would be able to train $G$. From this perspective $D$ can be though of as the *loss function* for training the generator, and in this setup we *learn* the loss function as well instead of having a pre-defined loss function such as $L_2$ distance between real and fake samples.

The training of GAN networks has become more stable because of recent work on particular architectures and loss functions for training. However, there remains lot of open questions regarding the convergence of the training procedure and if the training can indeed converge to the point where the data distribution can be modelled, and finally what type of distributions can the GAN model.

**Conditional generative networks** Our discussion till now has involved modelling the data set distribution $P(x)$. If the samples are available with their associated labels, it is also possible to use a GAN to model the distribution $P(x|y)$. This is implemented by providing as input the target label, $y$, or some encoding of it, as input to the generator along with the noise vector [70]. The task of the generator is to now map the noise to a sample that resembles the real training data *and* belongs to the category $y$.

Implementation can take the form of first creating a one-hot encoding of the label vector, then transforming this one-hot encoding to a dense representation through an embedding network. In practice this learnt embedding is more necessary for the discriminator than for the generator. In some cases, for the generator it is sufficient to concatenate the noise vector and the one-hot label vector. In more recent architectures, conditioning has been enforced by class conditional

`batch_norm` layers [12], where the gain and bias of the batch norm is dependent on the conditioning class.

**Convolutional generative networks** In the context of image generation, GAN modelled as convolutional architectures have been designed for which the training is relatively stable. See Figure 2.7 for samples generated from a BigGAN model. DCGAN [77] proposed a convolutional GAN architecture which has been used extensively. DCGAN generator is composed of *transposed convolutional* layers, which start with noise vector and produce an image sample. The convolutional structure enables it to model local patterns and compose them into complex images. The discriminator network is the inverse of the generator, and uses standard convolutional layers, and resembles a classification network.

DCGAN created the template of having a generator as a kind of 'reversed' classifier. The building blocks were then replaced by residual blocks, which are ubiquitous in image classification tasks. Architectures such as BigGAN [4] have trained large GANs which produce very compelling images. Transposed convolutions can learn interactions in a local neighbourhood, and across larger spatial areas indirectly. To model interactions among features across larger spatial areas, the concept of non-local operations can be used [105], and more specifically self-attention [110].

Figure 2.7 – Example images generated from a BigGAN model. Images taken from [4].

# Chapter 3

# Hierarchical Classification

In the classification task, a discriminative classifier is trained on a data set of labelled pairs $(x, y)$, where $x \in \mathbb{R}^D$ is a data sample (a vector of $D$ dimensions) and $y$ is the corresponding target category vector. A classifier is trained on such a data set, with a suitable loss function. After training the classifier produces a output vector $y'$ for a given sample $x'$. The output of a classifier is often treated as a distribution over the categories, which means that given a sample the classifier gives as output the probability of the test sample belonging to a particular target category. In the case of neural networks this is often done by applying a `softmax` operation to the final layer output.

Consider a situation where we have a semantic hierarchy tree of categories. In this tree, each node represents a *coarse* category and its child nodes are its corresponding *fine* nodes. For example, at the fine level we can have categories such as {car, truck, bike}, and at a coarse level they can be organised into a {vehicle} category. It is expensive to annotate data and annotation at a fine level may require both time and expertise. In contrast it will be cheaper and faster to annotate data at a coarser level. We are interested in the question of how to adapt a classifier trained at a coarse level, to classify over the finer categories, as and when such data becomes available. We can imagine that initially we have a classifier trained at a coarse level. Later we get access to the same data set annotated at the finer level and want to train a classifier and use the information contained in the older classifier trained at the coarser level. This is one instance where a trained system needs to adapt to changes in the data distribution. For this chapter the setting is that the label distribution changes but the data samples remain the same.

More specifically in this chapter we take classifiers trained at different levels of the label tree and investigate if we can utilise the relationship among the categories to improve the performance of the classifiers. This is achieved by enforcing the consistency of output probability distributions at different levels of a semantic

hierarchy of the target categories. Adapting the output distributions according to this requirement results in improvement in generalisation error. This can also be thought of as a way to inject domain knowledge during inference with the classifiers. The experiments in this chapter are performed using deep convolutional networks trained on an image classification task. Since this technique is concerned with consistency of probabilities at different semantic levels, it can work with any machine learning model that produces an output probability distribution over categories, such as SVM with Platt normalisation.

## 3.1 Method

In this section we define our problem setting and present the classifier adaptation method. We also define the kind of classifier used for the image tasks that we validate our method on.

### 3.1.1 Problem Setting

We work in a standard multi-class classification setting. For each data point, $x \in \mathcal{X}$,the goal is to assign a class label $y \in \mathcal{Y} = \{l_1, ..., l_N\}$. The assignment of labels for each input depends on the output of a classifier, $f : \mathcal{X} \to \mathcal{Y}$, as $f(x) = \arg\max_y f_y(x)$. We consider a probabilistic classifier, which predicts a probability distribution over $\mathcal{Y}$. This is often obtained by applying a `softmax` transformation to the final model output.

We consider a setting with two sets of class (or category) labels corresponding to a two-level hierarchy with $N_c$ *coarse* labels $\mathcal{Y}_c = \{c_1, ..., c_{N_c}\}$ and $N_f$ *fine* labels $\mathcal{Y}_f = \{f_1, ..., f_{N_f}\}$. The hierarchical relation is defined by a function $s : \mathcal{Y}_f \to \mathcal{Y}_c$, specifying that the coarse category with label $c_{s(j)}$ is the super-category of the fine category with label $f_j$.

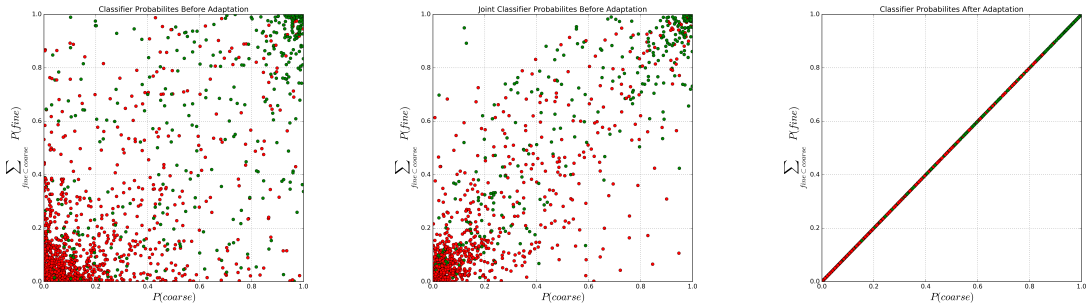
Given different classifier models, each of which are trained on different levels of the hierarchy of categories, we would like to get an improvement in the classifier performances at minimum additional cost. The goal is to aid the classifier in improving its predictions by augmenting the information contained in its parameters with the additional information from the hierarchy of categories.

Suppose we are given one classifier trained on $N_c$ classes and another classifier which is trained on $N_f$ classes. Each of the $N_f$ classes are *fine* classes of the $N_c$ *coarse* classes. Any given test sample will be assigned a probability of being correct by both classifiers. From this setting we make two observations:

1. It is intuitive to reason that the probability of being correct at the *coarse* level should be greater than the probability of being correct at the *fine* level.

This is due to the fact that the classifier has to discriminate among fewer classes at the coarse class level than at the fine class level.

2. The probability assigned at the coarse class level is distributed among the fine classes which belong to that particular coarse class. This is possible if similar the compositionality of features can determine similar categories, which share the same coarse super-category.




(a) (b) (c)

Figure 3.1 – Distribution of probabilities for coarse categories (`X-axis`) and total probability assigned to corresponding fine categories (`Y-axis`). (a): Separately trained classifiers, (b): joint training of one classifier on fine and coarse categories, (c): Probabilities adjusted to enforce that $P(coarse) = \sum P(fine)$.

The probability values output by a coarse and fine classifier are shown in Figure 3.1a. We consider a single coarse category (class 1 from CIFAR-20. See § 3.2). The X-axis shows the probability assigned to test samples for belonging to the coarse category. The Y-axis shows the total probability assigned to the corresponding fine categories, for the same test sample. Green dots denote positive data points which actually belong to this coarse category. If a sample is assigned high probability of belonging to the coarse category and also of belonging to one of the fine categories, it will be near the top right of the figure. This would mean that classifiers trained on the data set find the task of labelling the sample with both fine and coarse labels to be of similar difficulty. We see that there are several green dots which are either near the bottom right or the top left. This means that for for some samples, the classifier has high confidence in predicting the coarse category but not the corresponding fine category. The reverse case is true for points near the top left. We see that the hierarchical information is not consistent among the predictions of the classifiers trained at different category levels. In the current situation, for some samples, the fine category is predicted with high confidence but not the coarse category. Intuitively, we expect the prediction of the coarse category to

be an easier task since the number of categories over which the classifier has to make a prediction are smaller. We want to enforce this consistency requirement during the testing phase of the classifiers. We will see in the next sections that adjusting the probability distribution of the classifier to enforce this condition leads to improvements in generalisation error. The probability output distribution after adjusting the classifier predictions is shown in Figure 3.1c.

### 3.1.2 Classifier adaptation and inference

In Figure 3.1a, we see that the previous observations do not hold for all data points. We want to enforce the condition that the output probability that a classifier has for a coarse category should be equal to the total probability that a classifier has for all the corresponding fine categories. Formally, this is denoted as:

$$\forall i \in \{1, ..., N_c\} \; P(c_i|x) = \sum_{s(j)=i} P(f_j|x) \tag{3.1}$$

This state is depicted in Figure 3.1c. For all data samples, Equation 3.1 will be satisfied if we map them to the diagonal. This procedure is outlined in Algorithm 1. We start with the initial values assigned to the coarse category and the sum of the probability assigned to the corresponding fine categories. The prediction probability will be consistent if both probability values are mapped to the average of the coarse probability and sum of fine probabilities. First the coarse probability is adjusted by linearly interpolating by a factor $\alpha$ away from the original value to the target value (Line 7). Then we adjust the probability value for each of the fine categories of the current coarse category. The target probability of each fine category is computed in proportion to its original probability value (Line 11). Then a similar linear interpolation is carried out.

In order to control the degree to which we enforce Equation 3.1, an interpolation factor, $\alpha$ is used, as mentioned above. In Algorithm 1, the term $\alpha$, is present inside the `for` loop. This term allows us to linearly interpolate between the original probabilities and control the *degree* of adjustment that we want to carry out. In other words, it controls the extent of transformation from Figure 3.1 left to right. A value of $\alpha = 1$ is the situation shown in Figure 3.1c and a value of $\alpha = 0$ is the situation in Figure 3.1a. It is to be noted that $\alpha = 0$ means that we are not performing any adjustment. The value of this hyper-parameter can be adjusted during inference and allows flexibility. $\alpha$, can be tuned using a validation set to obtain the best results. Since, we modify the output probabilities based on the test sample, we term this procedure the *classifier adaptation* step.

Given the trained models, inference is done by using Algorithm 1. For each test sample, first we obtain the probabilities from the coarse and fine classifiers. These

**Algorithm 1:** Classifier adaptation

**Input:** Probabilities from coarse and fine classifiers
**Output:** Adapted classification probabilities, adjusted_coarse[i], adjusted_fine[i]

coarse[i], fine[i] //Lists of coarse and fine classes;
fine_coarse[j] // Coarse class of fine classes $j$;
P_coarse[i], P_fine[j] // Probability of coarse and fine classes;
Q_coarse[i] = $\sum_{fine\_coarse[j]=i}$ P_fine[j] // sum of probability of coarse classes of class $i$;
**for** $i$ = *1, #coarse* **do**
    target_prob = (P_coarse[i] + Q_coarse[i]) / 2;
    adjusted_coarse[i] =
    $(1-\alpha)\times$ P_coarse[i] + $\alpha\times$ target_prob ;
    **if** *Q_coarse[i] != 0* **then**
        **for** $fine\_coarse[j] = i$ **do**
            fine_target_prob =
            (P_fine[j] × target_prob[i]) / Q_coarse[i] ;
            adjusted_fine[j] =
            $(1-\alpha)\times$ P_fine[j] + $\alpha\times$ fine_target_prob;
        **end**
    **else**
        adjusted_fine[j] =
        adjusted_coarse[i] / ($\sum_{fine\_coarse[j]=i} 1$)
    **end**
**end**

probabilities are input to Algorithm 1. The output of the algorithm are adjusted probabilities. The final prediction corresponds to the class having the maximum probability after the adjustment. The cost involved in doing this step is linear in the number of the super-classes.

Note that the adjustment of probabilities is done according to the hierarchy of all the labels. It is not the case that only the true label of the sample is being used. This procedure is a *pre-prediction* step and hence can be used with any model which outputs a probability distribution over target categories.

### 3.1.3 Deep convolutional classifiers

We evaluate the proposed method on an image classification task. We choose deep convolutional network (CNN) as our classifier, since these models have very

good performance on image tasks. Good model architectures for the CNN are ResNet [37] and DenseNet [45]. These architectures serve as strong baselines and are the current architecture of choice in the community.

A ResNet model is composed of several `residual blocks`. Each residual block contains a certain number of convolution layers, followed by a `batchnorm` [48] layer and a non-linearity, such as ReLU. A residual block takes as input, feature maps of dimension $d \times d$ and outputs feature maps of dimension $d' \times d'$. Based on their feature map dimensions, convolutional layers of residual (and many other) networks can be organised into a number of computation stages (typically three). At the output of each stage (or block), the feature maps are down sampled using a pooling layer, and the number of feature maps is (typically) doubled in the following stage. This shown in Figure 3.2a. Figure 3.2b shows a simplified view while including the `softmax` operator and negative log-likelihood loss function used during the training phase. The architecture of DenseNet [45] follows a similar pattern, with the only difference being that each of each stage is called a "dense block", within which a layer $L$ takes as input the output feature maps of all previous layers $\{1...L-1\}$.

### 3.1.4 Merged architectures

In our discussion so far, we have needed to use separate classifiers trained at the coarse and fine category levels. This leads to increase in memory requirements and possibly slower inference. These can be reduced by introducing some degree of parameter sharing among the two different models. We do this by merging some of the layers among the coarse and fine models. The intuition is that several shared features will be useful for both the coarse and fine categories. This may lead to faster and more robust feature learning as the training phase will benefit from gradient signals from the two loss functions.

The granularity of the merge is at the level of the residual blocks. It has been shown that inside each block, features are refined. New features are computed only when the spatial dimensions change at the end of each block [30, 104]. Moreover, merging (or splitting) layers inside a residual block is non-trivial (especially in the case of DenseNet). Taking all of these into account, it is reasonable to have the shared parameters at the level of the blocks and not at a finer level. We explore merging strategies after each of the 3 computation stages. After each of the stages, a `maxpool` layer is used to aggregate all the features that have been computed so far. This serves as a point in the computation pipeline until which point the layers should be shared. We denote the usual training setup by `separate` (Figure 3.2c). In this case, two models are trained, one for the coarse and one for the fine categories. For the case of merged models, training is done for both category levels simultaneously. We explore the following variants of merged models:

- `merge-0` Fine and coarse models are trained independently but both receive the same input and the loss is summed (Figure 3.2c). Though the networks in Figure 3.2c and 3.2d are different, they should theoretically lead to exactly the same training and predictions.

- `merge-1`, `merge-2`, `merge-3` are depicted in Figure 3.2e, 3.2f, 3.2g. The common blocks in the layers are closer to the input. Each model has a separate fully connected classification layer for the coarse and fine categories. `merge-3` corresponds to a joint training where all layers are shared and only the fully connected layers are separate.

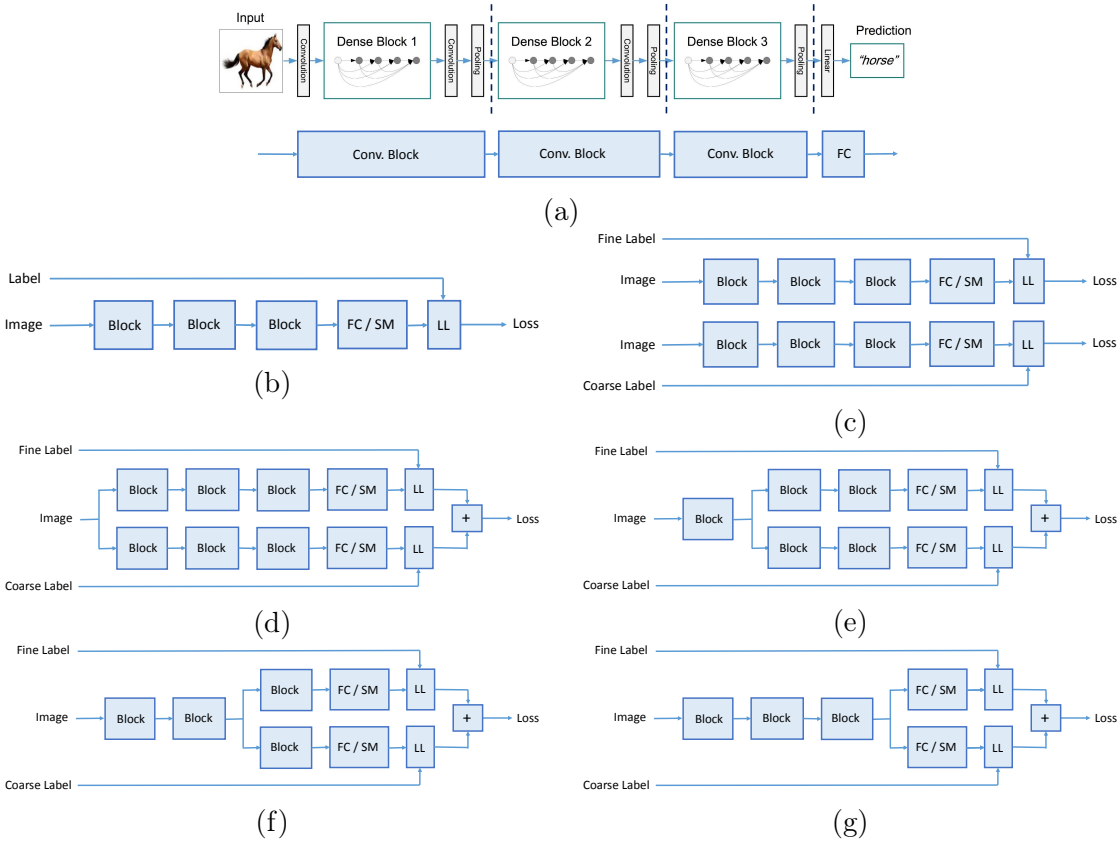



Figure 3.2 – `Merge` architectures: (a) Standard computation stage setup in current deep convolutional networks (Image taken from [45]; (b) simplified diagram including the `softmax` and loss functions for the training phase; (c) coarse and fine classification with separate networks; (d) merge of the loss function only; (e)-(g) progressive merging of the convolution blocks.

## 3.2 Experiments

**Data set** We use CIFAR-100 data set [53] for evaluating the classifier probability adjustment procedure. This data set contains 50 000 training images and 10 000 test images. The images are organised into two levels of semantic categories which we utilise. These images are split into 100 mutually exclusive classes that we shall consider as the fine classes and refer to as CIFAR-100. These 100 classes are grouped into 20 coarse classes that we shall refer to as CIFAR-20. Each coarse class has exactly 5 fine classes.

**Classifier training** Three deep convolutional neural network (CNN) architectures were used: ALL-CNN-C [97], ResNet-164 [37], DenseNet-100 [45]. We re-trained all these models on CIFAR-100 and CIFAR-20 ourselves. We additionally trained them on the fused set of concepts "CIFAR-120" that directly compute both the coarse and fine class probabilities at once. In this latter case, the `softmax` output layer is split in two parts, corresponding to the 100 fine and 20 coarse classes. The results obtained for CIFAR-100 are slightly different from those reported in the corresponding papers (see § A). Standard data augmentation was used during training, which is horizontal flips with 0.5 probability and random crops of $32 \times 32$ after padding each training image with 4 pixels on each side. The hyper-parameters and learning rate schedules were set according to the reported values for each architecture in their corresponding papers.

**Separate model training** The three different network architectures were trained on both CIFAR-100 and CIFAR-20 separately. After training, we applied our method as outlined in Algorithm 1. In all cases we observe that the probability adjustment leads to an improvement by lowering the error rate on the test images. The error rates obtained with no ($\alpha = 0$) and full ($\alpha = 1$) adjustment are shown in Table 3.1 ("Separate" column). It is important to point out that we obtain this gain in performance without any re-training and the adjustment time is negligible compared to the network training times. Figure 3.3 (blue dots) shows the change in error with respect to the degree of our adaptation scheme. We observe that on increasing the value of $\alpha$, we obtain a decrease of the error. Hence, steadily going from the situation depicted in Figure 3.1a to the one depicted in Figure 3.1c is better in the case of predictions from separate models.

**Joint model training** The three different network architectures were trained on CIFAR-120 (joint training). After training, the output probabilities are adjusted according to Algorithm 1. We see that the error rate for coarse category classification improves but for fine category classification it is worse. The error

rates obtained with no ($\alpha = 0$) and full ($\alpha = 1$) adjustment are shown in Table 3.1 ("Joint" column). Figure 3.3 (green squares) shows the change in error with respect to the degree of our adaptation scheme. We observe that on increasing the value of $\alpha$, we obtain a decrease (and also a corresponding increase) of the error rate.

**Model ensembles** Exploiting the hierarchical relationship among the labels by using the output from two classifiers has some similarity with ensemble learning. The main difference with the usual ensembling setup is that instead of double training with either twice the coarse or fine categories, the two models have a different view of the data as they are trained with different labels, the coarse and fine category levels. We do a contrast experiment by training two models and then averaging the predicted probabilities. The "ensemble" predictions are better than the adjusted ones, either from separate or from joint training, in all cases except for the All-CNN-C network on the coarse categories. However, t is also possible to do adjustment from the ensemble predictions and it once again always leads to improved performance as shown in Figure 3.3 (orange diamonds).

| Model | Separate | Joint | Ensemble |
|---|---|---|---|
| CIFAR-20 | | | |
| All-CNN-C initial | 20.72 | 19.74 | 19.78 |
| All-CNN-C adjusted | 18.42 | 18.48 | 17.58 |
| ResNet-164 initial | 15.25 | 14.62 | 13.76 |
| ResNet-164 adjusted | 12.77 | 13.94 | 11.72 |
| DenseNet-100 initial | 13.74 | 13.16 | 12.11 |
| DenseNet-100 adjusted | 11.32 | 12.31 | 10.22 |
| CIFAR-100 | | | |
| All-CNN-C initial | 30.65 | 29.63 | 29.43 |
| All-CNN-C adjusted | 29.64 | 29.94 | 28.48 |
| ResNet-164 initial | 23.32 | 23.35 | 21.09 |
| ResNet-164 adjusted | 22.21 | 23.45 | 20.48 |
| DenseNet-100 initial | 21.07 | 21.33 | 18.89 |
| DenseNet-100 adjusted | 19.98 | 21.47 | 18.25 |

Table 3.1 – CIFAR-100 error rates before and after adjustment of probabilities.

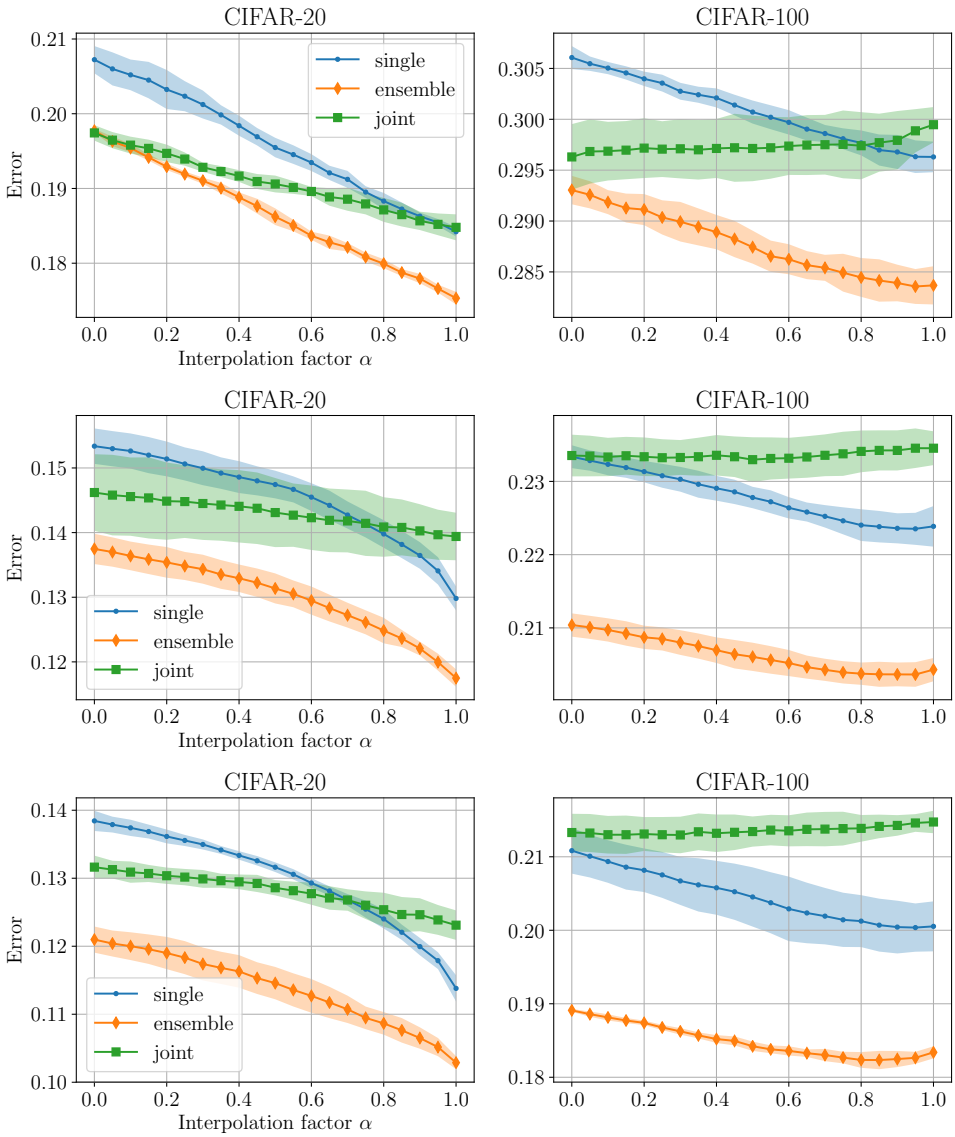



Figure 3.3 – Classification error with respect to interpolation. Left: Coarse categories (CIFAR-20), Right: Fine categories(CIFAR-100). For each series, we report mean ± standard deviation.

### 3.2.1 Merged architecture results

We trained the five different merged architectures described in Figure 3.2c-3.2g four times for ResNet-164 and once for DenseNet-100. We then applied classifier adaptation while varying the $\alpha$ parameter from 0 to 2 and we plot the top-1 error rate for both coarse and fine categories as a function of $\alpha$ for all combinations.

Results are presented in Figure 3.4 for ResNet-164 and in Figure 3.6 for DenseNet-100. In Figure 3.4, plots are given for ensemble learning with ensembles of different sizes: `ensemble-n` corresponds to the averaging of the training of $n$ different instances of the same architecture (ensemble-1 corresponds to the training of a single instance).

For ResNet-164, using the four different training, it is possible to get a rough estimate of the standard deviations around the mean value. In Figure 3.4, the solid line denotes the mean and the shaded regions denote one standard deviation around the mean. For the ensemble-4 case, it is not possible to directly estimate the standard deviation so the one obtained from ensemble-3 under the same conditions is used as an estimation.

**Analysis of ResNet results** From Figure 3.4, we can make the following observations:

- The statistical differences between the various merged architectures is low since there is a significant overlap between the shaded areas and these are only at one standard deviation. In particular, the `separate` and `merge-0` are slightly different though they should theoretically be identical. Still, the difference becomes sharper when considering the `ensemble-2` and `ensemble-4` versions and the evolution of the performance according to $\alpha$ is likely to be significant.

- For both the CIFAR-20 and CIFAR-100, the optimum value for the $\alpha$ parameter is quite close to 1 as expected and classifier adaptation leads to a significant reduction in error rate, confirming our hypothesis.

- Regarding CIFAR-20, the performances obtained only from the probability prediction for the fine categories ($\alpha = 2$) are significantly better than the performances obtained only from the probability prediction for the coarse categories ($\alpha = 0$) but the performances obtained by combining both ($\alpha = 1$) are even better.

- Regarding CIFAR-100, the performances obtained combining probability predictions for both coarse and fine categories ($\alpha = 1$) are significantly better than the performances obtained only from the probability prediction for the fine categories ($\alpha = 0$).

- Regarding the different merged architectures, the intermediate ones, `merge-1` and `merge-2`, appear to have equivalent performances and to be both better than the fully split or fully merged ones.

- Ensemble learning provides a very significant gain as this has already been observed in simple training (without considering label hierarchy) but classifier adaptation provides an additional gain for all number of instances, especially for the coarse categories, even though the gain tends to decrease with the number of instances.

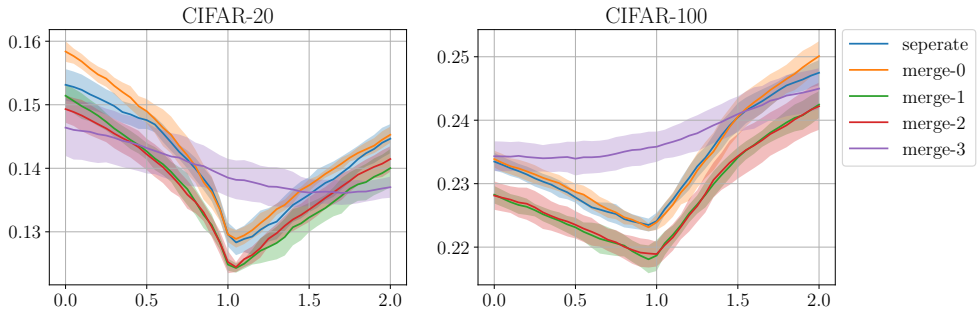
**Analysis of DenseNet results** From Figure 3.6, we can make the following observations:

- The differences between the curves are probably not all significant though their individual shapes might be.
- As expected, the DenseNet error rates are lower than that of ResNet.
- Regarding the merged architecture and the classifier adaptation aspects, the results are comparable with those obtained with ResNet but not fully consistent. `merge-2` has now a behaviour closer to `merge-3` (joint training) than to `merge-1` (as for ResNet). This indicates that the split between the fine and coarse network paths should be kept deeper than in the ResNet case maybe because of the long-distance shortcuts in the dense networks.

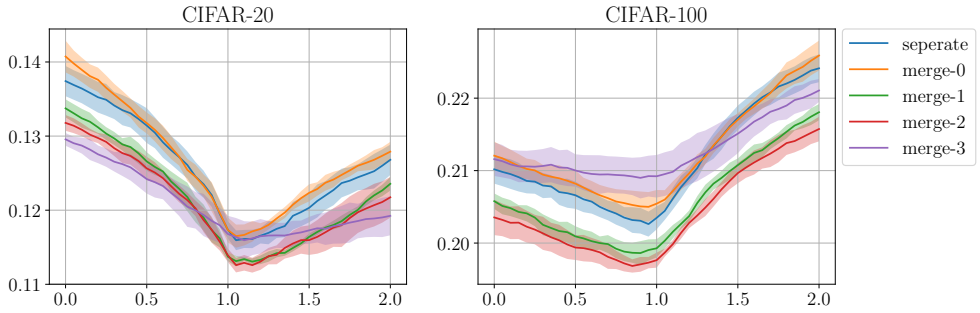
## 3.3 Discussion on related work

Most current work involves fine tuning a trained classifier to achieve good performance on a new data set that is different from the one on which the training was done. There is little work that addresses the issue of adapting a classifier to situations where the data distribution remains the same but the label distribution changes.

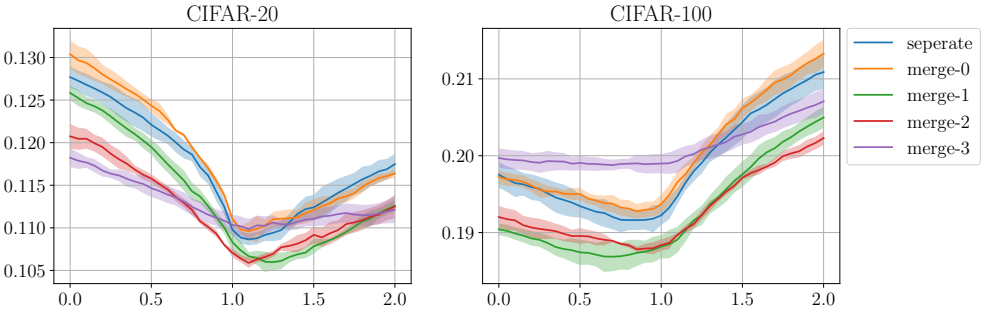
Royer et al.[85] proposes a probabilistic framework to adapt the predictions of a classifier, based on the sequence of images that are presented to the classifier at test time. Their objective is to take advantage of the fact that in real life scenarios, test images are not presented randomly but have correlations. For examples, after an image of a tropical tree is shown, it is unlikely that the next test image will be of a coniferous tree covered in snow. The classifier adaptation strategy in [85] uses the signal present in the sequence of past test images to alter prediction probabilities of future images, which is achieved by modifying *class prior* probabilities. Their work is concentrated on a flat distribution of categories and does not utilise any hierarchical relationships. In our case, we are concerned with adjusting and correcting our classifier by taking into account hierarchical relationships among object categories. This adjustment is by taking into consideration just one given




(a) `ensemble-1` for the five merged architectures




(b) `ensemble-2` for the five merged architectures




(c) `ensemble-4` for the five merged architectures

Figure 3.4 – Error rates (Y-axis) for the different merged `ResNet-164` based architectures with respect to the interpolation factor $\alpha$ (X-axis). `ensemble-n` represents the error rate after averaging the predictions from $n$ models.

test sample and not the previously seen test samples. The correction they do also takes into account the feedback from the prediction, that is whether the classifier was correct or not. In our case our adaptation algorithm does not take into account the feedback because the relationship among categories is fixed and cannot be changed.

In the context of using a hierarchy of categories, a closely related work is that

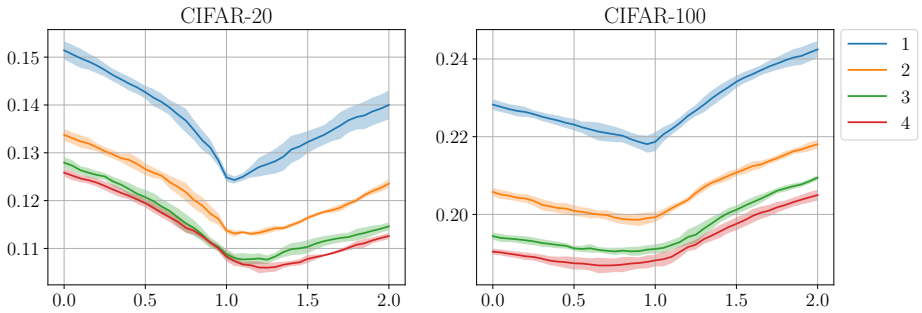



Figure 3.5 – Comparison of `ensemble-1` to `ensemble-4` for the `ResNet-164` `merge-1` architecture.

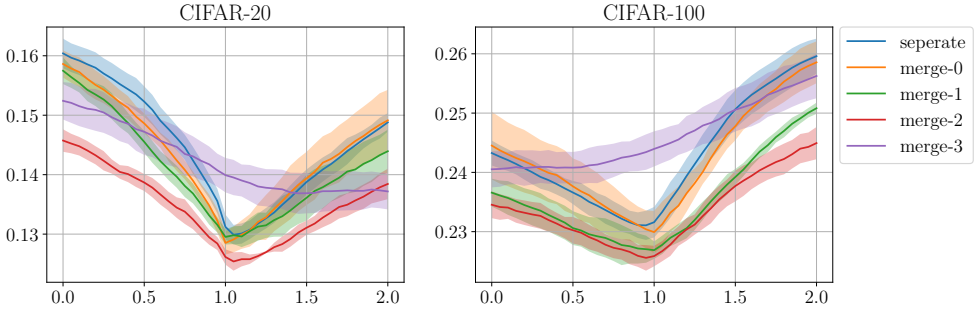



Figure 3.6 – Error rates (Y-axis) for the different merged `DenseNet-100` based architectures with respect to the interpolation factor $\alpha$ (X-axis). Results are from a single run of `ensemble-1`.

of Jia et al.[49]. In this work, the authors propose an algorithm that discovers a sub-tree of categories. The predictions of the classifier are then restricted to this sub-tree, out of entire tree of categories that the classifier is trained on. Restricting the prediction to some categories improves generalisation of the classifier since it has to discriminate over fewer categories. The system identifies the sub-tree on being given a few test images. The goal is to have a single classifier, that is trained over a large number of categories, which adapts to the sequence of test images. This is similar to the previously discussed work of [85]. However, they work in a setting involving a single classifier with high capacity, trained on large amounts of data. In our setting, we aim to 'transfer knowledge' between two distinct classifiers which were trained separately. The only information we use during inference is the given hierarchy of categories and a single test sample.

Hierarchical relationships have been used in the context of image retrieval tasks in [13]. In this work, the authors use the semantic relationships among categories as a 'ranking' function to determine related imaged when a query image is pre-

sented. This work is in the case of retrieval, as compared to classification in our case. However, it highlights that if the semantic relationships are available it can be utilised to improve the performance of the system. Another work that uses hierarchy of concepts is YOLO9000 [81]. The training of YOLO9000 is done by fusing the concepts from ILSVRC [87] and MS-COCO [63], where the fusion is done using the common category nodes from each data set. Further at prediction time, the `softmax` operation is done child nodes of the same parent. This utilised the semantic hierarchy present. YOLO900 makes fixed predictions though, and its output is not adaptive in any way. The semantic tree is used in a similar way to our method but there is no dynamic adjustment after the probability values have been calculated. Finally, HD-CNN [108] utilises a hierarchy of categories by training classifiers at each level of the hierarchy tree. The levels denote progressively harder categories from the root to the leaf of the tree, and this tree is learnt from data. At inference there is no explicit adjust of the output but there is a possibility to execute only some of the trained classifiers. This constrains the output to one part of the tree, similar to the case of YOLO9000.

## 3.4 Conclusion

In this chapter we introduced a classifier adaptation procedure to adjust the predictions of the classifier based on the consistency of probability distributions among a hierarchy of categories. We showed that enforcing the constraint among probability distributions among different levels of this hierarchy resulted in improvement of the classifier performance. Next we showed that instead of training classifiers at different levels of the label hierarchy, it is possible to have a shared model to some extent, which can predict a probability distribution at both levels. Currently we have considered two levels of the categories but it is possible to extend this further over several levels. Extending to several levels will require changing our adaptation procedure to make sure that the probabilities are consistent along each path from the root to a leaf, and across each level. Further we want to explore if it is possible to *learn* the semantic hierarchy if it is not available. It will be interesting to see what hierarchy we can recover and how the specifics of the hierarchy affect the final adjusted probabilities.

We explored in this chapter, a scenario where the labels of a data set change, more specifically the labels are split into sub-categories to enable fine-grained classification. This is one example of situation where a trained classifier needs to be adapted with a change in the data distribution. Here we considered only a change in the label distribution with no additional data points being presented to the classifier. In the next chapter we investigate a more general case where new data

sets is presented to the classifier sequentially.

Our experiments with merged architectures were motivated by the need to reduce the memory needed for multiple models. We saw that in this specific context we can have some sharing of parameters. However, the sharing led to separate predictions since by design we need different outputs at different semantic levels. We saw that in particular `merge-1` and `merge-2` had better performance as compared to the separate training and all layers shared (`merge-3`). This indicates that an architecture with parallel branches may lead to superior performance. In Chapter 5 we will investigate this observation in detail in a systematic manner. We will see where the gain in performance comes from and explore various architectural choices with respect to training with parallel branches.

# Chapter 4

# Continual Learning

Machine learning models are usually trained on a data set and then deployed in an evaluation mode. During deployment we have an underlying assumption that the data distribution is not changing. However, it is not entirely unlikely that the distribution does change. For example we can have an animal classification system which can classify ten different animals. After some time when we have data available for two new animals, we would like to extend the classification system so that it can classify the new animals as well. In the case of neural network based models, this is challenging to achieve and the answer regarding how to do it is far from trivial.

The sub-field which deals with questions such as these is referred to by several names such as: continual learning [82], lifelong learning [102], or incremental learning. In the current discussion, we will use the term *continual learning*, and in the context of this work we assume that all three terms are equivalent. Recently, several approaches have been proposed to enable continual learning in neural networks. The experiments and results from these works show that, for most approaches, learning 'continually' leads to a worse performance as compared to learning the tasks simultaneously. Here, we take a step back and ask the question of how good is the representation that is learned in a continual learning scenario as compared to training on all of the tasks at once. The differences in the representations learnt are a likely cause of the difference in performance and may offer pointers to how to reduce the gap in performance.

We focus on the usage of *replay buffers* for continual learning in neural networks. The implementation of a replay buffer in the form of generative models has been proposed and used in the literature [83, 93]. We use replay buffers implemented as Generative Adversarial Networks (GAN) [29]. In particular we investigate the current state of GAN models with respect to their ability to generate samples from which good discriminative classifiers can be trained. We see that current GAN models are not yet able to completely capture the diversity and qual-

ity of real training samples. Next, we analyse a continual learning scenario where we are allowed access to old task data. Our aim is to compare the representation learnt for several tasks in a continual learning scenario, with the representation learnt when training on the same tasks simultaneously.

## 4.1 Context, definition and notation

In this chapter we explore the idea of continual learning in feed forward neural networks for supervised learning. Examples are fully connected networks and convolutional networks. We consider cases where these networks have been used in image classification tasks. In principle, the methods described and discussed, can be applied with any feed forward network and a suitable differentiable loss function. However, these have not been considered explicitly here.

Lifelong learning, incremental learning, continual learning are all similar sounding and meaning terms and hence there is a need to define each of them specifically. For now we consider all of them to be equivalent and use the term *continual learning* in this chapter. We define continual learning as follows:

- There are a set of tasks $\mathbf{T}$ and each task has its own dataset $D_t$. For each $t$ we associate a performance measure $A_t$
- Each $\mathbf{D_t}$ is presented to our model $M$ sequentially.
- When a new $D_{t+1}$ is presented to $M$, the learning algorithm must update $M$ such that $\forall t < t+1$ $A_t$ must not be negatively affected while $A_{t+1}$ must be maximised.

When training neural networks with a differentiable loss function and stochastic gradient descent, we assume that the data is *i.i.d*, which means that the selection of each data sample is independent and the data samples are from the same distribution. However, in continual learning that is not the case since tasks can vary greatly.

Let us make explicit the difference between continual learning, transfer learning and multi-task learning [6]. Continual learning is different from **transfer learning** where once $D_{t+1}$ is observed we are not concerned with any previous $D_t$. The primary objective of transfer learning is to re-use a trained model in case the new task does not have sufficient training samples and (or) speed up training for the new task. It is different from **domain adaptation** where the nature of $D_t$ changes. Take for example a model trained on images of human faces, captured with a standard camera. Domain adaptation will be required if the model is to be deployed on a cartoon version of the human images (like a sketch version of the faces). Continual learning differs from **multi-task learning** due to the sequential

presentation of $D_t$. In the case of multi-task learning the model is optimised to learn on all the tasks at once, while in continual learning the tasks are presented sequentially.

Some example scenarios which might be modelled as a continual learning scenario are:

- Addition of new categories
- Refining existing categories to sub categories
- Additional training data for existing categories
- New task for the system
- Extending trained classifiers over additional categories

**Catastrophic Forgetting** Many machine learning systems do well on the task that they are trained on. This is especially true for modern neural network systems. However, when the system is trained on tasks *sequentially*, it may forget about the older tasks. This phenomenon is known as *catastrophic forgetting* [78, 69]. In neural networks, the primary obstacle to incremental learning is catastrophic forgetting. Due to this reason, a number of works concentrate on reducing catastrophic forgetting as a means to enable continual learning.

The reason for catastrophic forgetting in neural nets is due to the backpropagation gradient computation algorithm [86]. At each forward pass we compute the loss for the data batch through an error function. Then during the backward pass, the gradient of this loss with respect to each parameter is computed. An optimisation step is carried out using this gradient. We can summarise this through the following equation, where $f(x, w)$ is the output of a model $f$, for a data batch $x, y$. The model has parameters $w$. $\mathcal{L}$ is the loss function to measure the error between the model prediction and ground truth $y$, and $\eta$ is the learning rate:

$$w_t = w_{t-1} - \eta \nabla_w \mathcal{L}(y, f(x; w_{t-1})) \tag{4.1}$$

We note that there is only the notion of a data batch and no notion of what task this data batch belongs to. Hence, during training the algorithm is only concerned with minimising the loss for the current batch. This is why catastrophic forgetting (or interference) occurs. If we are training for a new task $T_j$ after an older task $T_i$, the optimisation algorithm will update the parameters such that they are optimal for $T_j$ and completely ignore performance on $T_i$. To prevent this forgetting, we will see that various methods propose to preserve the memory of $T_i$ through several techniques such as replaying old training data, preserving responses of output units

corresponding to old tasks or preventing parameter values to change too much from a configuration suitable for $T_i$.

In the context of modern neural networks, [27] has performed empirical comparison among different variations with respect to catastrophic forgetting. Their experiments set the precedent for the evaluation of different continual learning methods. In particular, the paper proposed to evaluate these methods in three categories: forgetting in similar domains, forgetting in different domain, input reformatting tasks. The components evaluated include regularisation strategy, training algorithm and activation function. We will see that most evaluations of more recent continual learning methods follow this methodology.

## 4.2 Continual learning approaches

Continual learning approaches can be grouped into several categories and we explore some of these options here, along with their related advantages and drawbacks. For some approaches any particular category might not fully describe it or it might be that some methods do not fall into a particular category. Based on current work, we propose a grouping as logical as possible. The important point is to understand that there are several differing approaches to reduce catastrophic forgetting.

### 4.2.1 Memory based approaches to incremental learning

In this category of techniques for incremental learning, the core idea is to store data which will help to avoid degradation of the model's performance on the older tasks. The memory can be used to store data samples, feature vectors or any encoding of the original data. In rehearsal based techniques discussed below, training data of older tasks in stored. In gradient episodic memory (discussed in the next section), gradients of data with respect to the loss are stored. The interesting case is when the entire data set is not to be stored. Then we can ask the question: how many data samples to store, and which data samples to store?

**Gradient episodic memory** [65] motivates the idea of continual learning by introducing a scenario where a machine learning model has to learn from data samples which might belong to separate tasks. The general way of learning a model is by applying the ERM principle where we learn a model by minimising a loss function. One of the assumptions that make ERM valid in practice is that training data samples are *iid*, which means that each data sample is *independent* of each other, and that they are sampled from the same *identical* distribution. However, in a continual learning scenario the model will need to learn on tasks

sequentially, and during each training step, the *iid* assumption does not hold. This leads to issues in applying ERM in an incremental learning scenario, where the data is not *iid*, the data of current tasks may adversely affect older learnt tasks (catastrophic interference). On a positive note, if the data of the current task is related to a previous task, transfer learning can help speed up current learning and also maybe help the older task. Dependencies between tasks can be captured through a suitable encoding of the tasks. This task description can be used to model the relationships among various tasks.

The setup for experiments is that all training samples are of the form $(x, t, y)$, where $t$ is a task descriptor. The samples are locally *iid* for each task. In this work, integer task descriptors have been used. However, potentially rich task descriptors can be used to enable transfer learning across tasks. They define metrics to evaluate their models, namely: average accuracy, forward transfer, and backward transfer. Given the current task $t$, forward transfer measures how an older task $t_i$ affects the performance of $t$. Backward transfer measures how after training for task $t$, the performance of an older task $t_i$ is affected.

Gradient episodic memory essentially consists of storing samples of a task $t$ in a memory $M_t$. The total memory $M$ comprises the $M_t$ memory cells for each of the tasks. There can be a fixed budget of the memory $M$ which means that $M_t$ keeps changing as the number of tasks increase. Currently, they do not use any sample selection procedure and simple store the last $|M_t|$ samples for each task. They mention building *coresets* as a strategy for saving samples in the memory.

If the training is done to minimise the loss on the current task and the loss on $M$ there would be overfitting to samples in $M$. If we enforce a loss to prevent change in the output (for example by distillation loss as in LwF [61], iCARL [80]) we remove the possibility of positive backward transfer since we are trying to prevent changes to the old outputs. Instead we can use an inequality which says that the loss on $M$ can be decreased but not increased. If a decrease in loss is allowed, then we have the possibility of positive backward transfer. In practice this is realised by a dot product between the loss gradient for the current task sample and the loss gradient for older samples (for the current parameter state).

**Rehearsal mechanisms** One way of preventing forgetting of older tasks is to present the old task data to the network when it is being trained on a newer task. This is termed as *rehearsal*. An extensive discussion on rehearsal and various techniques was presented in [83, 78]. In rehearsal mechanisms a separate memory is used to store the real data or a proxy for the old data, such as a generative model. Rehearsal mechanisms without using real data has been proposed in [83, 2] and is termed pseudo rehearsal. These methods obtain pseudo input target pairs by feeding random noise matrices through a trained network. They posit

that the output vector obtained by feeding random noise is sufficient to preserve the knowledge. It has not been tested on modern convolutional networks trained on images.

In Deep Generative Replay (DGR) [93], the real training data is not used directly. Instead they learn a generative model of the training data. When new tasks arrive, a generative model is used to provide pseudo training data for the older tasks. Since generative models as of today are not perfect, the degree to which older tasks will be preserved is limited by the quality of the generative model. This quality refers to both the perceptual quality and diversity. Recent work has shown that GAN architectures cannot sufficiently replicate the quality and diversity of the true data set distribution [91].

We also have the question of why do we not store the real training data itself. There can be several reasons for this. One reason is privacy where we are able to share only the model trained on this data, which can be thought of as a featurised and compressed version of the training data. The data is restricted for sharing because of privacy concerns, legal concerns, etc. Another reason is that the rate of data production is growing faster than the rate at which data storage hardware is advancing. We will soon not have the capacity to store all of the data. In this case the generative model can serve as a loss compressed representation of the data, from which we are able to obtain a comparable performance. We also note that there is no provision for extending the capacity of the network. This implies that there is a finite number of tasks that the network can solve and there needs to be a mechanism to increase the capacity.

### 4.2.2 Structural regularisation approaches

Structural regularisation approaches aim to find a set of parameter weights that are optimal for the tasks being considered by the model, without any modifications to the architecture. This is accomplished by modifying the loss function to take into consideration the performance of older tasks. One of the first methods was first proposed in [40], where there are two types of weights operating at different time scales to preserve old tasks, and also to adapt quickly to new tasks. More recently, in Learning without Forgetting (LwF) [61] the loss function enforces that the output corresponding to older tasks do not change while the new task is being learnt. In Elastic Weight Consolidation (EWC) [52] there is a per parameter regularisation to prevent weights that are important for the older task from changing too much. The general idea of this category of approaches is shown graphically in Figure 4.1.

We have mentioned before that one of the reasons for catastrophic interference is that when gradients are calculated using backpropagation there is no notion of which task is being optimised for. Conceptor aided backprop (CAB) [38] proposes

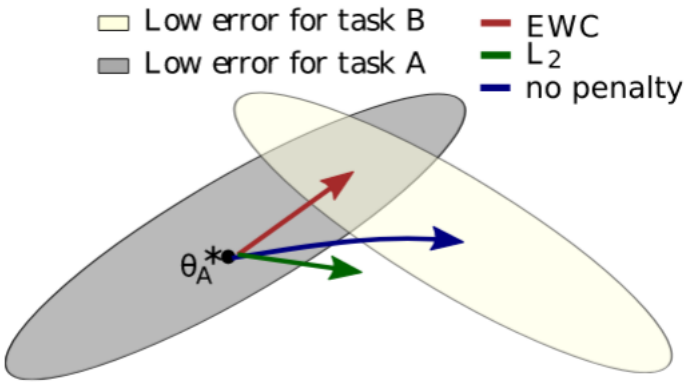



Figure 4.1 – The shaded areas show the set of parameter values that are optimal for two tasks A and B. Structural regularisation techniques aim to find the intersection of these two sets. Figure taken from [52].

to take into account previous tasks when the gradients are being calculated. This is done by checking how the current task gradient would modify the weights learned on previous tasks. The procedure modifies gradients such that the weight space of older tasks is preserved.

This category of approaches are inherently limited as a general continual learning strategy because there is no scope for increasing model capacity. This means that after a certain number of tasks are added, we will reach a saturation point where the average generalisation performance will suffer. One can argue that we can start with a grossly over parameterised model and then we can keep on adding tasks. This would still imply that only a finite number of tasks can be added and we would be forced to use a inefficient model (in terms of number of parameters) until the full quota of tasks is reached. It does not present a general adaptive framework for continual learning.

### 4.2.3 Architectural approaches

Architectural approaches work under the assumption that different tasks require separate sets of parameters. The spectrum includes models which have all parameters separated and also models where there a set of parameters common to tasks and a separate set of task specific parameters. The main idea is that the architecture of the model is explicitly modified.

On one end of the spectrum we have Progressive Networks [88], where a complete network is added as new tasks are presented to the system. At the start, we have a single task and a network with $L$ layers. When a new task is presented

all $L$ layers are duplicated. Now the model has two columns of $L$ layers. The second column has horizontal connections per layer with the previous column(s). The new column is trained with data of the new task and all old columns are kept fixed. By design, there is no catastrophic forgetting of the old tasks. Note that the number of columns grows linearly with the number of tasks. A related method is ExpertGate [1], where there is a network for each task. The parameters of new task networks are initialised with the parameters of the 'closest' old task network. This distance between tasks is measured through an under-complete autoencoder, where each autoencoder is a compact representation of the respective task. The autoencoders are also used at test time to choose which task network to evaluate. ExpertGate is the only method mentioned here that does not need an 'oracle' to identify which task the current test sample belongs to.

On the other end we have Dynamically Expandable Networks [109] where the architecture is modified only as needed, depending on the new tasks. The summary of the algorithm is: identify task specific parameters at each layer for a new task and train these parameters; if performance is below a pre-defined threshold add new parameters and train; finally to preserve performance of old tasks, check if parameters specific to older tasks have changed greater than a given $\delta$, if yes duplicate these parameters. Overall, this approach is more parameter efficient as compared to Progressive Networks.

### 4.2.4 Universal representations

Universal representations aim to solve several tasks by explicitly learning features specific to each of the tasks. As several tasks are introduced to the model we can aim to learn representations useful for solving these tasks explicitly, in contrast to developing methods that try to adapt an existing representation for task $T_i$ for another task $T_j$. In principle explicitly learning universal representations is a more flexible approach that should lead to better generalisation performance across a distribution of tasks.

Residual Adapters [79] have shown that given a CNN architecture for different image classification tasks, the only thing that needs to be specific to each task are the BatchNorm layers. This means that we can have a universal representation by employing a base CNN architecture in which the convolutional layers are task agnostic and all the Batch Norm layers are task specific. [32] employ a similar idea where they use depth-wise separable convolutions followed by point wise convolutions at each layer. The depth-wise convolutions are task specific and the point wise convolutions are common to all tasks. Empirically they show that they achieve a similar level of performance with a lower parameter cost. Deep Adaptation Networks(DAN) [84] also have the aim to solve several tasks using a single model. DAN consists of a backbone network and controller modules to select

| Method | #Parameters | Mean score | Decathlon Score [79] |
|---|---|---|---|
| Residual Apapters [79] | 2 | 77.17 | 2643 |
| DAN [84] | 2.17 | 77.01 | 2851 |
| Piggyback [68] | 1.28 | 76.60 | 2838 |
| Depthwise Conv [32] | 1 | 77.82 | 3507 |

Table 4.1 – Comparison of methods aiming to learn universal representations.

among tasks. The controller module specifies for each task a linear combination of filters at each convolution layer. This approach is likely to work well when all tasks are related to other other. In other cases a linear combination of filters might prove to be restrictive.

Going further, Piggyback [68] proposes to have task specific parameters at all layers and not just the BatchNorm layers. Again, we have a backbone network from which task specific weights are identified. Piggyback identifies redundant weights for a particular task and prunes these for that task. The pruned weights are available to be used for other tasks. In this manner, at each layer of the network we end up with sparse weights that are task specific. The sparse combination of weights can be identified with task specific masks which control how each layer is used for a particular task. The end result is another way of obtaining a universal representation for the given tasks.

Each of these methods have a task specific prediction layer. There is also the assumption of an oracle which identifies the task for a given test sample. There is some discussion in [79] on predicting the task identity but the other works do not mention this. A comparison of these methods are in Table 4.1.

### 4.2.5 A note on the different approaches

The different approaches discussed have various pros and cons with respect to growing memory, execution times, etc. Techniques which have a fixed architecture have an inherent limit to the number of tasks that they can accommodate because they have a specific number of parameters. However, architecture expanding approaches can theoretically accommodate a large number of tasks, at the cost of increasing memory.

Replay buffers can enable several tasks in one network but need to be sequentially trained. Intuitively replay buffers seem to be more flexible than structural regularisation approaches because there is greater freedom in changing the neural network weights to optimise for several tasks. This is why we investigate replay buffer based methods further.

Inference over several tasks is also a topic of further discussion and research.

For most of the methods discussed, the task specific output layers are chosen to make the final prediction. However, if the given task is not known there needs to be a mechanism to decide which task a given test sample belongs to. The knowledge of the task greatly changes the final performance of the system, as the chance for errors increase when we have a common output layer over all tasks and the task of the test sample is not known.

## 4.3 Generative adversarial network as a replay buffer

We now turn our attention to using replay buffers as a means to prevent catastrophic forgetting in neural networks. Replay buffers store data, or some representation of the data, belonging to older tasks. This data is used in conjunction with data from new tasks while training a combined model. There is a discussion in using replay buffers in fully connected networks in [83]. More recently, Deep Generative Replay (DGR) [93] proposes to use a GAN as a replay buffer. The use of a GAN is motivated by the fact that in the present continual learning setting, we are not allowed (or may not be able) to store the actual data. The basic idea of DGR is use a generative model of old task data to prevent forgetting. Every time a new task is to be learned, a training data batch is created by combining the old data samples from the GAN and current task data set. This is depicted in Figure 4.2.

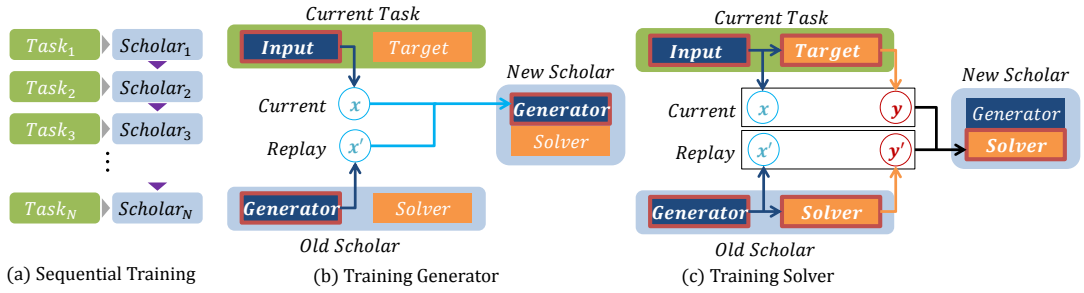



Figure 4.2 – Overview of Deep Generative Replay (DGR). Figure taken from [93]

DGR raises two questions. First, since a GAN is used to remember old task data, the degree to which the performance of old tasks is preserved is determined by the quality of the GAN model. In this section we study how good is the training of a classifier from GAN samples as compared to real data. Next, in DGR we see that the training procedure for the classifier is the same at every task, which means that the overall training time for all tasks is no better than training on all tasks at once. We will study strategies for training on tasks sequentially in § 4.4.

### 4.3.1 How good are current GAN models?

Recent GAN models, such as BigGAN [4], are able to produce samples which are quite hard to distinguish from real data. Generating 'realistic' samples does not necessarily correlate to being able to train good discriminative classifiers, which is our primary objective. In order to be as good as the real training data set, the generative model needs to be able to reproduce both the inter category and intra category diversity. In addition to having enough diversity, the samples should be visually meaningful. The ability to do this has several applications such as data compression and data privacy protection. A generative model can be thought of as a compressed version of the real data. In a scenario where it is not possible to store or share the real data, we can instead have, and share, the compressed generative model. In several applications, such as in the medical domain, data sharing may not be possible due to factors such as protecting data of patients. In these case we can instead share the generative model, and others can reuse this data for other applications, without compromising privacy.

We concentrate on comparing the samples from BigGAN [4], a state of the art GAN architecture at the time of writing, to real data samples. This comparison is carried out from the perspective of training a good discriminative classifier. In order to compare generated samples to real data, several metrics have been proposed such as Inception Score (IS) [90] and Fréchet Inception Distance (FID) [39]. These metrics aim to characterise how 'realism' of the generated samples. Hence we compare the validation error of a classifier after training on samples from the GAN, and after training on the real data set. Indeed we also investigate if good IS and FID scores correlate with classifiers having low validation error rates.

In the next sections, we first show that training with samples directly from a GAN does not lead to the best training of a classifier, even though the samples look very realistic. We introduce a filtering mechanism to remove bad samples which helps in obtaining higher quality samples, leading to training of better classifiers. Since a generative model has limited capacity, it may not be able to capture the complete data distribution. In order to cover more of the data distribution, we propose to sample the training data from multiple GAN models.

### 4.3.2 Sample filtering

In order to train our classifier, we generate data batches by sampling from a conditional GAN model. This means that the images are generated from a noise vector and label on which the generating process is conditioned. In theory all the generated image should belong to the corresponding conditioning label but in practice this is not always the case. Additionally, even if the generated image belongs to the correct category it may not be discriminative enough and may be

missing some features. These cases arise because the generative model is not a perfect model of the training data distribution. We propose to use a pre-trained classifier, trained on the same training data as the GAN, to filter out such samples. This is a general approach that can scale to larger data sets and categories. The procedure for selecting a sample image from the GAN that will be used to train a classifier is described in Algorithm 2.

We can choose different threshold values for the prediction probability obtained from the pre-trained classifier. This ensures that we get samples that belong to the correct category and are also informative enough to train good classifiers. We discuss the results of training a classifier and the effect of choosing different probability thresholds in §4.3.4. We will see that choosing a proper threshold becomes more important when the number of target categories increase.

**Algorithm 2:** Selecting samples from the generator

**input :** $C$, $G$, $\theta_G$, $Class_{pre}$, thresh
**output:** List of samples

$sample\_list = []$;
$y \sim \mathcal{U}(0, C)$;
$z \sim \mathcal{N}(0, 1)$;
$x \sim G(z, y; \theta_G)$;
$p = Class_{pre}(x)$;
**if** $argmax(p) == y$ *and* $p[y] > thresh$ **then**
  add $x$ to $sample\_list$
**end**

### 4.3.3 Multiple GAN sampling

Our training data sets are that of natural images which have significant variation among categories and also within each category. We expect that this implicit distribution is difficult for current generative models to capture because the models have a limited number of parameters , our training algorithms are not perfect, and we are not guaranteed that the training will converge to the global minima of the loss function. The current literature suggests that GAN models suffer from mode collapse which leads to the model being unable to capture all aspects of the data distribution.

We can train multiple GAN models on the same data set to cover more of this distribution, where each GAN model might be able to model different parts of the distribution. We hypothesise that this might be possible due to different starting initialisation of each of the GAN models, and non-convex loss surface

during the training procedure of the GAN model. During training of the classifier from synthetic samples, we can construct our data batch by sampling from these multiple GAN models. Samples of categories coming from different GAN models can help us to cover the data distribution in a more effective manner. Experiments and results are discussed in §4.3.4.

### 4.3.4 Experiments

**Datasets and models** We use the CIFAR-10 and CIFAR-100 [53] data sets for our experiments. These data sets consist of 50 000 training images divided into 10 and 100 categories respectively. The test data set comprises 10 000 images. Each image is $32 \times 32$ pixels and 3 channel RGB. The GAN model is trained using these data sets. In the following discussion we will refer to the original data set as $D_R$. We also construct a synthetic data set by sampling images from a trained GAN model. We will refer to the synthetic data set as $D_G$.

We use the BigGAN [4] architecture as our GAN model. The conditional version is used to enable sampling images per category. The conditioning is achieved via conditional batch normalisation layers. The training of the GAN is done according to the original paper with related settings of the hyper-parameters. For the classifier we use a 18 layer ResNet [35] model with 11M parameters because it provides a good trade off between performance and training time.

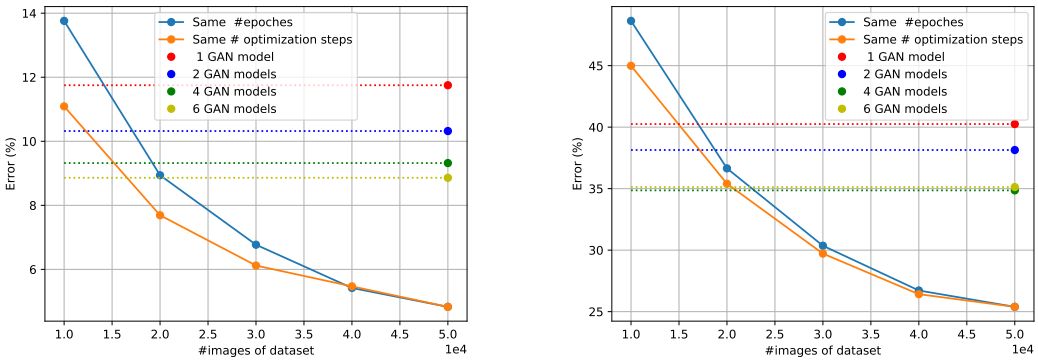
**Classifier performance comparison between real and generated data** The classifier is trained on both $D_R$ and $D_G$ is done for 156 250 steps (200 epochs with batch size 64). In the case of $D_R$ this means iterating over the same samples multiple times as is done usually. In the case of $D_G$ a batch of samples is obtained from the trained GAN model. We expect in the ideal case to have different image samples throughout the training, however in reality we are limited by the capacity of the GAN model with respect to diversity of the data.

The blue curve depicts training on different subset sizes for 200 epochs. This means that in case of smaller data set sizes, we have less number of optimisation steps. The orange curve shows training on each subset for the same number of optimisation steps. We see that in the case of less samples, training longer reduces the classification error rate substantially.

The same classifier is trained on GAN samples, for the same number of optimisation steps and following the same learning rate decay schedule. Each batch of samples for training the classifier are generated according to Algorithm 2. The error rate for these classifiers is comparable to training on $1 \times 10^4$ and $1.5 \times 10^4$ real data samples in CIFAR-10 and CIFAR-100 respectively (see horizontal lines in Figure 4.3, which are quite far away from the lowest points of the curves). This

means that training on GAN samples lead to worse performance as compared to training on the full data set. This indicates that the GAN is either not able to produce samples similar to the real data or the intra class diversity is not as much as in the real data set. Visual inspections of samples show that the generated samples do indeed look quite similar to the real images (Figure 4.6).

**A compression view of GAN** A trained GAN can be viewed as a compressed version of our training data, where the learned weights are the compressed representation of our original data bits. We are interested in compressing the nature of the distribution from which the training data has been sampled, rather than the explicit training data samples themselves. To measure the degree of compression, as before we train a classifier on samples from a GAN and measure the prediction error on a held out validation set. The lower bound for the error that the classifier can achieve is equal to the error that it achieves after training on all of the real training data. We compare training on the GAN samples with varying amounts of real training data to see how the test error evolves with varying amounts of data, and at which point training on GAN samples is comparable with real training data (see Figure 4.3a and Figure 4.3b, detailed results are in Table 4.2). We see that as of now, training on GAN samples is comparable to training on around $10 \times 10^3$ real data samples. The size of the GAN model is around 15MB as compared to the 30MB for the $10 \times 10^3$ samples. This shows that with a GAN we are able to obtain good compression without compromising performance, when the number of data samples is less. However it also highlights that current GAN samples are quite far away from entirely modelling the real training data distribution.




(a) CIFAR-10. (b) CIFAR-100.

Figure 4.3 – Training a classifier on real data compared with training on GAN samples

**Effect of filtering threshold** A GAN sample is used for training the classifier if it belongs to the correct category and if the output probability is above a pre-defined threshold. In Figure 4.4 we plot the relationship between output probability of the pre-trained classifier and the final validation accuracy of the classifier trained on GAN samples. We observe that the value of the threshold is not significant in the case of CIFAR-10. This implies that the pre-trained classifier is quite confident *if* it makes a correct prediction. The reason for this could be that the GAN is able to generate very good images and the classifier is therefore confident in its predictions. In our experiments this implies that if the pre-trained classifier makes a correct prediction in most cases we can use the corresponding sample in our synthetic training set. We need to study this further for different data sets with varying number of target classes.

However, in the case of CIFAR-100, the probability threshold is important and increasing the threshold leads to a reduction of the classifier error from 39.35% to 37.62%. The threshold makes sure that only good quality image are used for training the classifier. On the other hand, a high value of threshold can reduce the diversity of generated samples which might lead to worse generalisation for the classifier.

**Issues with threshold based on prediction probability** Filtering based on probability values raises several questions. If we are filtering out samples based on a probability threshold, one implication is that we are choosing only easily classified samples. The pre-trained classifier might have biases during training which can imply that it finds some particular images easy to classify. One step to reduce this bias towards 'easy' examples is to add noise to the predicted probability of the pre-trained classifier. The addition of noise will reduce the bias by allowing some examples to be used for training, which did not have a probability greater than the threshold. There is another issue which is regarding the calibration of output probabilities. Currently we do not take this into account but for future work this should be handled. The output probabilities should be calibrated before comparing with the threshold. This is important since we use recent powerful neural network models and these are known to have probability calibration issues [31].

**Filtering threshold analysis** At this filtering stage we can analyse which categories are filtered out more. This would be one indication that the GAN model has trouble to capture the distribution for these categories. We can further analyse the predictions and features of the pre-trained classifier for the generated samples. Similar predictions and features would indicate the repetition of samples and lack of diversity. This can be done both at the inter and intra class level. More details

will be explained in the next section.

In the case of CIFAR-10, most bad images are removed because of incorrect label (green bar in Figure 4.5a). This is one reason why the threshold value does not have too much effect. On the other hand, in Figure 4.5b we see that in the case of CIFAR-100, the number of bad images which is filtered out by the threshold value, increases remarkably. There are a few categories for which it is easy to generate images belonging to the correct category, while for the majority of categories we need to sample repeatedly to get images of the correct category (green bar in Figure 4.5b). One concern here can be that for these hard categories, the intra category diversity is reduced when we sample repeatedly since the generator model is inherently of limited capacity. This is a subject of future work, to investigate and improve diversity for hard categories. We visualise several samples before and after filtering out the bad samples in Figure 4.6a and Figure 4.6b.

In both data sets we see that the use of the pre-trained classifier is useful for removing samples that do not belong to the correct category. The effect of the probability threshold selection becomes more important as the number of categories increases. It has a significant effect in improving the validation error of the classifier trained on generated samples.

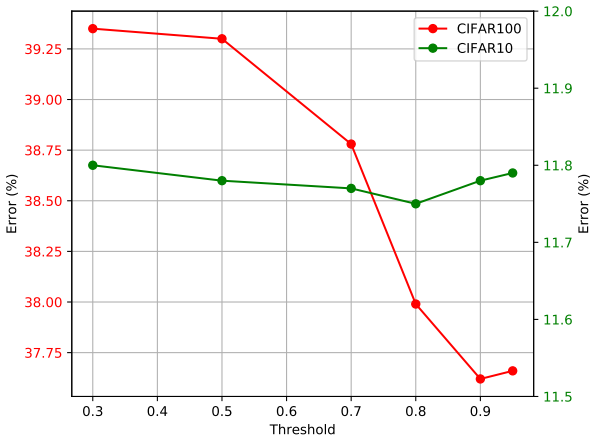
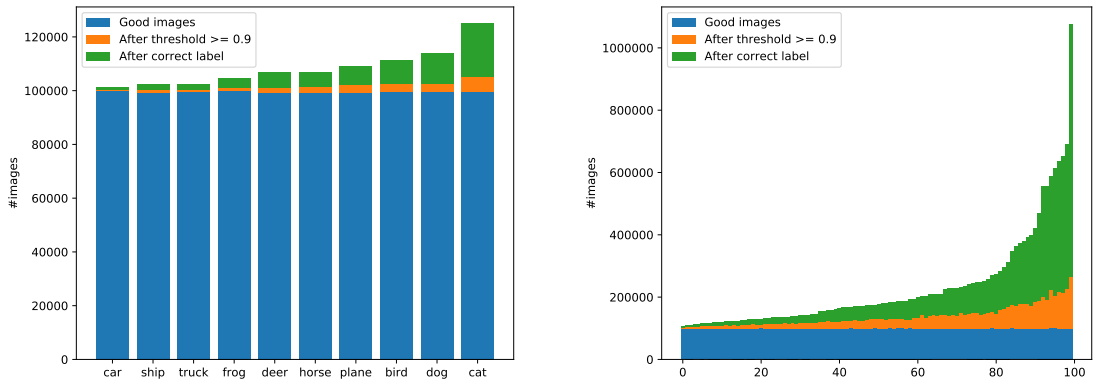



Figure 4.4 – Effect of threshold on classifier training from GAN samples in CIFAR-100.

**Multiple GAN sampling** We see from Figure 4.5 that the distribution of some categories are harder to capture for the GAN model. For some categories, we must do repeated sampling in order to get acceptable samples that are not rejected by the pre-trained classifier. We introduced the idea and intuition behind using multiple GAN models for training the classifier in §4.3.3. During training of the

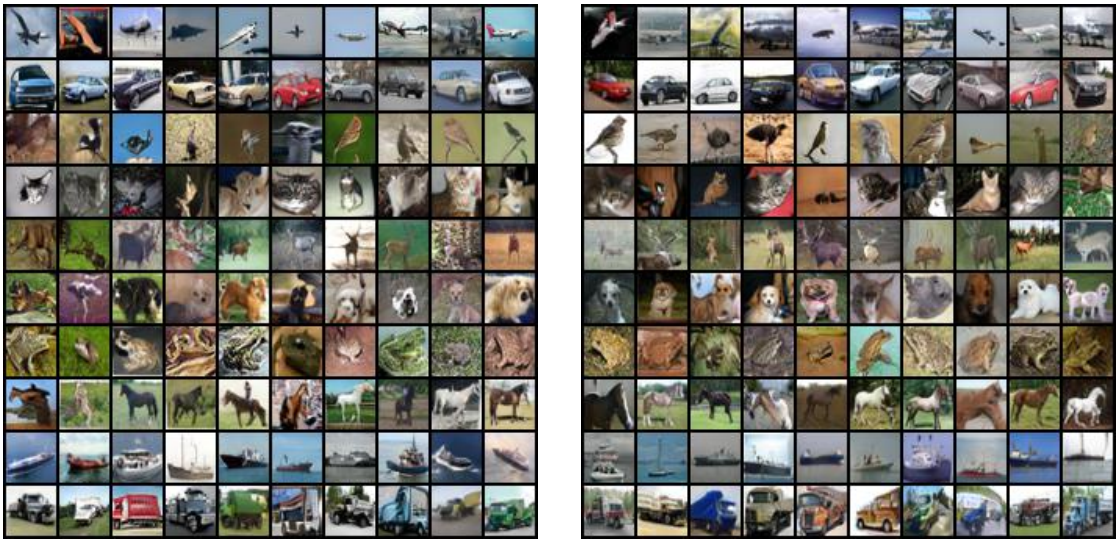



(a) CIFAR-10 (b) CIFAR-100

Figure 4.5 – Sample generation distribution according to categories

classifier from synthetic samples, we can construct our data set by sampling from the multiple GAN models which can help us to cover the data distribution in a more effective manner. In Table 4.2 we see that sampling training images from two, four and six GAN models leads to a reduction in classification test error from 11.75% to 8.86% for CIFAR-10 and from 40.25% to 35.13% in CIFAR-100. In the case where we use 4 GAN models for generating the samples, the error in test samples decreases by around 1% and 3% in CIFAR-10 and CIFAR-100 respectively. This technique is particularly useful when dealing with large number of categories. In spite of not reaching to the error of a classifier which is trained with $30 \times 10^3$ real data samples, the gap is significantly reduced while also requiring less storage bits as compared to real data. Another point to note is that FID score is improved in all cases, which is one indication that multiple GAN models help to improve the diversity of generated samples.

We see that although the change of IS and FID with different number of GAN models is small, the error on testing samples of a classifier have a greater reduction. From Table 4.2, in CIFAR-10, the value of IS and FID is in the range of 8.62 to 8.58 and from 6.84 to 6.47 respectively, however, the error is decreased by 3%. This indicates that IS and FID scores are not the best way to evaluate a GAN when the objective is to use the GAN for downstream tasks such as training another classifier.



(a) Before filtering (b) After filtering

Figure 4.6 – CIFAR-10 samples before and after applying the filtering procedure

## 4.4 Experimental analysis of continual learning with replay buffer

We saw in the previous section that a GAN model is not yet able to replace the real training data samples such that a good classifier can be trained. We proposed methods to close the gap in classifier training between real and generated samples but there is still a gap. If a GAN model is used as a replay buffer in a continual learning scenario, where tasks are presented sequentially, this gap is likely to be even larger.

The methods discussed in §4.2 have considered the sequential presentation of tasks to a model. The problem setting enforces a constraint that data of older tasks are not available when training on a new task. In most of the reported experiments and results, this leads to sub-optimal performance when training on even moderately complex tasks such as image classification on the CIFAR [53] data sets. In most cases the methods are not able to completely alleviate forgetting on older tasks when the number of tasks added are greater than 2. Additionally, the final performance on all tasks is worse than when training on all tasks together. If we are in a continual learning setting, we want the training procedure to provide gains in one or more scenarios: training time, better generalisation, backward and forward transfer among tasks, etc.

If we relax the data constraint by allowing access to all of the training data corresponding to older tasks, it is still not obvious how the training should be

Table 4.2 – GAN and classifier metrics

| #GAN models | Size (MB) | IS (↑) | FID (↓) | Error (%) |
|---|---|---|---|---|
| CIFAR-10 | | | | |
| 1 | 16.6 | 8.62±0.13 | 6.84±0.69 | 11.75±0.23 |
| 2 | 33.2 | 8.62±0.16 | 6.53±0.12 | 10.32±0.50 |
| 4 | 66.4 | 8.56±0.16 | 6.47±0.14 | 9.32±0.02 |
| 6 | 99.6 | 8.58±0.17 | 6.60±0.10 | 8.86±0.12 |
| Same #optimisation steps | | | | |
| 50K-real | 150 | 10.2 | - | 4.83 |
| 40K-real | 120 | | - | 5.47 |
| 30K-real | 90 | | - | 6.12 |
| 20K-real | 60 | | - | 7.69 |
| 10K-real | 30 | | - | 11.09 |
| Same #epochs | | | | |
| 50K-real | 150 | 10.2 | - | 4.83 |
| 40K-real | 120 | - | - | 5.42 |
| 30K-real | 90 | - | - | 6.77 |
| 20K-real | 60 | - | - | 8.94 |
| 10K-real | 30 | - | - | 13.76 |
| CIFAR-100 | | | | |
| 1 | 17.7 | 9.43±0.17 | 9.19±0.48 | 40.25±0.32 |
| 2 | 35.4 | 9.40±0.22 | 8.85±0.30 | 38.14±2.00 |
| 4 | 70.8 | 9.33±0.22 | 8.42±0.11 | 34.86±0.05 |
| 6 | 106.2 | 9.39±0.25 | 8.45±0.09 | 35.13±0.24 |
| Same #optimisation steps | | | | |
| 50k-real | 150 | 12.1 | - | 25.39 |
| 40k-real | 120 | - | - | 26.42 |
| 30k-real | 90 | - | - | 29.72 |
| 20k-real | 60 | - | - | 35.40 |
| 10k-real | 30 | - | - | 44.99 |
| Same #epochs | | | | |
| 50k-real | 150 | 12.1 | - | 25.39 |
| 40k-real | 120 | - | - | 26.72 |
| 30k-real | 90 | - | - | 30.36 |
| 20k-real | 60 | - | - | 36.65 |
| 10k-real | 30 | - | - | 48.64 |

done so as to be as efficient as possible and obtain the maximal gains. Our aim in this section, is to take a step back, and analyse the representation learnt when a model is trained on different tasks sequentially. If all of the data from older tasks is used, is the representation learnt different as compared to directly training on all of the data from all of the tasks? This is the main question that we seek to answer. Further using the data from older tasks can be thought of as sampling from a *perfect* generative model for all older task data. If we assume that we have access to such a model, how should we sample from it and how should the

training proceed? To answer these questions we design several continual learning scenarios: addition of a single category to a pre-trained model and addition of multiple categories to a pre-trained model. For each scenario we investigate the gains, if any, of incremental training, as compared to simultaneous training of tasks. To the best of our knowledge, this baseline scenario of continual learning, where all data from all tasks is available, has not been investigated before.

### 4.4.1 Training on disjoint parts of the data

In the first experiment, we demonstrate a simple case of catastrophic forgetting. We train a ResNet-20 [35] model on $40 \times 10^3$ samples of CIFAR-100 for 200 epochs with a step learning rate decay schedule. This model gets 34% error on the test set. Now we restart training with the remaining $10 \times 10^3$ samples and do not use the original $40 \times 10^3$ samples any more. We train for 200 epochs and follow the same learning rate decay schedule. After finishing training, this model obtains an error rate of 42.86% on the test set, which is much worse.

The two training sets contains examples from all categories but due to the sequential training method, the model has forgotten the knowledge from the first round of training and likely overfits to the 10K training samples of the second training round. As a control, if we train on all of the 50K training samples during the second stage, the model achieves an error rate of 31.30% which is consistent with training on all of the data at once. This is an expected result and the experiment serves to motivate the investigation of training procedure that can avoid these issues while bringing improvements in performance.

### 4.4.2 Evaluation metrics

The different measures are labelled $e_{k-l-m-n}$ and used as table headings in the following discussion:

$k$ : number of models (1 in non-incremental training, 10 otherwise);

$l$ : number of possible predictions during training (9 before incremental training, 10 otherwise);

$m$ : number of possible predictions during inference (9 before incremental training or restricted predictions, 10 otherwise), *equal to the number of classes over which the soft-max is computed*;

$n$ : number of input classes during testing (1 for the left-out class, 9 for the non-left-out classes, 10 otherwise).

Not all combinations are meaningful or useful. We consider:

- Training of a single model on all 10 classes:
  - $e_{1-10-10-10}$ : Standard average error over the 10 classes with 10 possible predictions. Baseline for comparison with $e_{10-10-10-10}$ (below).
  - $e_{1-10-10-9}$ : For each of the 10 9-class sub-sets corresponding to a 'left-out' class, average error over the 9 classes, with the 10 possible predictions *with a soft-max computed over all 10 classes*. Then, global average error over the 10 sub-sets. Baseline for comparison with $e_{10-10-10-9}$ (below). The global average error is the same as $e_{1-10-10-10}$ but we may look here at the per 'left-out' class error.
  - $e_{1-10-9-9}$ : For each of the 10 9-class sub-sets corresponding to a 'left-out' class, average error over the 9 classes, with 10 possible predictions *with a soft-max computed over the 9 'non-left-out' classes*. Then, global average error over the 10 sub-sets. Baseline for comparison with $e_{10-10-9-9}$ (below).
- Pre-Incremental training of 10 models on 9 classes with 9 possible predictions, each corresponding to a 'left-out' class:
  - $e_{10-9-9-9}$ : For each model, standard average error over their 9 classes with their 9 possible predictions. Then, global average error over the 10 models.
- Post-Incremental training of 10 models on 10 classes with 10 possible predictions, now including the previously 'left-out' one:
  - $e_{10-10-9-9}$ : For each model, average error over the original ('non-left-out') 9 classes *with a soft-max computed over the same 9 classes only*. Then, global average error over the 10 models. Difference with the previous one: what is lost due to the learning of the new class when this one is not in the possible predictions;
  - $e_{10-10-10-9}$ : For each model, average error over the original ('non-left-out') 9 classes *with a soft-max computed over all 10 classes*. Then, global average error over the 10 models. Difference with the previous one: what is lost due to the inclusion of the new class in the possible predictions;
  - $e_{10-10-10-1}$ : For each model, error over the added (previously 'left-out') class *with a soft-max computed over all 10 classes*. Then, global average error over the 10 models. This measure is for evaluating how well the new class is learned compared to the previously learned ones.

– $e_{10-10-10-10}$ : For each model, average error over all 10 classes *with a soft-max computed over all 10 classes*. Then, global average error over the 10 models. This measure is a 9:1 weighted average of $e_{10-10-10-9}$ and $e_{10-10-10-1}$.

### 4.4.3 Adding one category to nine trained categories

We start our analysis with the simple case of adding one new category to a model that is already trained on several categories. We expect that if a model is already trained on several categories it has already learnt features that should be useful for the new category. Given this hypothesis, we experiment with different strategies of training to add a single new category.

**Experimental setup** We train CNN models with ResNet-20 [35] and DenseNet-100-12-BC [45] architectures. Both of these architectures obtain decent performance compared to state-of-the-art, and are relatively fast to train. Since we need to do a large number of experiments, we choose these architectures to have a good balance between performance and training time. The training is first done on 9 (out of 10) categories of CIFAR-10. After this initial training has converged we add the remaining category. Data augmentation consists of random horizontal flips and random crops. The model is trained using SGD with a step learning rate decay schedule. The decay schedule is the same for both phases of training.

**Training strategies** At the time of adding a new category to a trained model, we have different choices for the number of training epochs, proportion of samples between old and new categories in a training batch ($\alpha$), and training with weight warmup. A summary of the results for training using ResNet are in Table 4.3 and 4.4. We can make the following observations:

- Table 4.3 shows the result of training ResNet on all categories. Training for 200 epochs with all classes gives an error rate of $7.73 \pm 0.20\%$. Further training with restart of another 200 epochs with a starting learning rate of 0.1 gives an error rate of $7.44 \pm 0.17$. These two results serve as baseline error rates for the next experiments.

- Table 4.4 shows the results for adding one category to a model trained on 9 categories. Each row corresponds to a different training method. All numbers are averaged over 10 training runs, with 1 category being left out at each of the runs.

- (200 + 200). Training on 9 categories for 200 epochs, followed by training on all 10 categories for another 200 epochs. The overall error is lower than

Table 4.3 – CIFAR-10 baseline error rates for ResNet and DenseNet

| Architecture | Single Train | Train half with restart | Train with restart |
|---|---|---|---|
| ResNet-20 | 7.73±0.20 | 7.78±0.15 | 7.44±0.17 |
| DenseNet-100-12 | 4.80±0.16 | 4.96±0.17 | 4.69±0.27 |

training on all categories, both single run and with restart (7.36 compared to 7.73 and 7.44). The reduction in error rate may not be statistically significant but this shows that at least in this case we do not lose any performance. However, we also have no gain in training time as we trained for the full 200 epochs.

- (200 + 100). Train for 100 epochs when adding the new category - This gives slightly worse results but better than the case of single run over all categories. This method reduces the training time of the new category by 100 epochs, with a slight loss in performance.

- (200 + warmup + 100). Freeze all convolution layer weights and train only the `fc` classification layer for 50 epochs. Then train all weights for another 100 epochs. The objective is to get the new added category to acceptable error as compared to the older categories. This would make the gradients on the 'same scale'. In practice the results are the same as training directly with 100 epochs.

- (200 + pseudo targets). Old categories are not used for training. We generate pseudo input-target pairs to preserve the model responses of old categories. This is done by giving as input 'random noise images' and recording the responses from output units corresponding to the older trained categories. These input output pairs are used as training data when the new category is to be added. However, results of this training strategy are almost the same as random error.

**Training batch composition** During the incremental training to add a new category, we can choose how to compose our training mini-batches. The experiments until now have used a uniform sampling of data samples from the old and new categories. Since gradients are calculated for each mini batch, the proportion of samples from the old and new categories will bias the optimisation procedure towards old and new tasks respectively. We analyse the effect of varying the expected proportion of samples from the new category $\alpha$. Since there are 10 categories, uniform sampling of data samples implies 0.1 probability of picking a sample from

Table 4.4 – ResNet incremental training on CIFAR-10. We start with a ResNet trained on 9 categories and then add the left out category.

| Train type | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|
| 200 + 200 | 7.73 | 6.76 | 7.34 | 7.58 | 7.36 |
| 200 + 100 | 7.73 | 7.00 | 7.60 | 8.08 | 7.65 |
| 200 + warmup(50) + 100 | 7.73 | 6.98 | 7.56 | 7.93 | 7.60 |
| 200 + pseudo targets | 7.73 | 84.63 | 90.51 | 50.97 | 86.56 |
| 200 + 200 Adam (0.01) | 12.3 | 11.83 | 12.67 | 13.54 | 12.76 |
| 200 + 200 Adam (0.001) | 9.20 | 10.70 | 11.58 | 12.20 | 11.64 |

the new category. We experiment with $\alpha = 0.1, 0.2, 0.4$. We train a DenseNet model in the same manner as the ResNet models with different values of $\alpha$ (Table 4.5). We see that higher values of $\alpha$ negatively impacts the final performance. Higher values of $\alpha$ mean that, in expectation, each batch has a greater proportion of samples from the new category. Greater proportion of samples from new task make the performance of older categories worse.

Table 4.5 – DenseNet training with varying batch proportion, $\alpha$. The weights of new classes are **are sampled from a standard normal**. First column: initial training on 9 classes; second and third columns: after incremental training for the 9 initial classes, on 9 classes and on 10 classes respectively; fourth column: after incremental training on the added class; fifth column: after incremental training for the 10 classes.

| $\alpha$ | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|
| 0.1 | 4.80 | 4.44 | 4.80 | 4.98 | 4.82 |
| 0.2 | 4.80 | 4.46 | 4.81 | 4.98 | 4.83 |
| 0.4 | 4.80 | 4.57 | 4.94 | 4.86 | 4.93 |

### 4.4.4 Adding 10 categories to 90 trained categories

In the previous section, we explored several strategies for adding one category to a pre-trained model. The results are not particularly indicative of any difference between training incrementally and training simultaneously. Going forward, we now analyse the scenario where we add multiple categories, by adding 10 new categories at a time. We use the CIFAR-100 [53] data set and the same ResNet and DenseNet architectures.

**Experimental setup** CIFAR-100 contains images belonging to 100 categories. These are grouped into 20 semantic super categories. We first train on 90 categories and then add the remaining 10. We choose the left-out categories in two ways: combining two of the given semantic groups, and random 10 categories. For both phases, training is done with SGD and step learning rate decay schedule.

**Initialisation of `fc` weights for new categories** The addition of new categories require the extension of the `fc` layer, where we have to add new weights for each new category. The choice of initial weights will have a significant impact on the training procedure. We hypothesise that this is more critical where we need to add a large number of categories. In particular, the initial gradients from these categories will be on a different scale as compared to the existing older categories. We validate the weight initialisation choice by comparing three strategies as shown in Table 4.6. Out of the three strategies the `kaiming_uniform` performs best. In the case of sampling new weights from the standard normal, the gap in error rates between simultaneous and incremental training is 1.20%.

Table 4.6 – Weight initialisation strategies for ResNet and DenseNet on CIFAR-100.

| Arch | Weight INIT | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|---|
| ResNet | 200 + 200 standard normal | 31.95 | 32.86 | 33.67 | 28.50 | 33.15 |
| ResNet | 200 + 200 nearest neighbour | 31.95 | 32.05 | 32.91 | 27.83 | 32.40 |
| ResNet | 200 + 200 kaiming uniform | 31.95 | 31.74 | 32.61 | 28.64 | 32.21 |
| DenseNet | 300 + 300 standard normal | 23.23 | 27.59 | 28.23 | 25.76 | 27.99 |
| DenseNet | 300 + 300 kaiming uniform | 23.23 | 23.48 | 23.96 | 22.70 | 23.84 |
| DenseNet | 300 + 150 kaiming uniform | 23.23 | 23.56 | 24.09 | 23.34 | 24.01 |

**Training strategies** A model is first trained on all 100 categories for 200 epochs and for 200 + 200 epochs with restart (see Table 4.7). These error rates serve as baselines for each of the different training strategies, the results of which are in Table 4.8. We can make the following observations:

- Training additional categories performs worse than training all categories together (32.21% vs. 31.86%). Considering that the 90 categories where essentially trained on twice, this error rate is quite far away from training with restart which obtains 31.16%. One possible reason for this can be that the starting weights for the training on all 100 categories is worse than a random initialisation. We will further analyse this in the next sections.
- In the case of the standard normal initialisation, a warmup step helps. During the warmup step, first we train only the `fc` layer for a few epochs, and

then continue training normally of all the layers. This makes sense since these warmup epochs bring the weights and hence the gradients to a scale similar to the weights for the older categories.

- ((R) 200 + 200). We note that which the grouping of categories also has an effect on the final result (last line of Table 4.8). In this case, we add categories by randomly picking from the entire set of categories. In all other cases, we add 10 categories which belong to a semantic group.

- (200 + 200 Distillation). We use a **distillation loss** [41] for the outputs of older categories, during the incremental training phase. In the incremental training phase, the loss function for older categories is the cross entropy loss for correct prediction of categories, and a KL divergence loss between output of the new model and the output of the old model. This helps the performance of older categories to be preserved. We have seen in the first row that the error rate for older categories is 32.61%, and using distillation loss reduces this to 31.59%.

- (200 + 100 Distillation). Since adding the distillation loss term helps the older categories, we can try a smaller number of epochs for the incremental training phase. In the case of starting with a high learning rate (0.1), we obtain similar results as compared to the full training of 200 epochs (31.73% and 31.33% respectively.) However when we train on 100 epochs by starting with a smaller learning rate (0.01) we get a worse result (32.77%). This implies that when training is started in the incremental step, a high initial learning rate is required to escape the minima obtained on all categories and find a good minima for the combined old and new categories.

- Results for adding 10 categories of CIFAR-100 to DenseNet are given in the lower half of Table 4.6. The weight initialisation is an important factor in determining the final combined performance. The results are consistent with the ResNet case.

- In conclusion, we can say that naive addition of categories actually leads to a drop in overall performance. This can be mitigated to some extent by using a distillation loss. In addition the distillation loss helps to shorten the incremental training phase.

### 4.4.5 Freezing blocks during incremental step

ResNet and DenseNet convolutions layers are arranged in blocks, where each block comprises several convolution layers. Each block operates at a fixed spatial resolution. We experiment on training different number of blocks, while keeping the rest

Table 4.7 – CIFAR-100 baseline errror rates for ResNet and DenseNet

| Architecture | Single Train | Train with restart |
|---|---|---|
| ResNet-20 SGD | 31.86±0.24 | 31.16±0.26 |
| ResNet-20 Adam | 34.02±0.13 | 33.03±0.56 |
| DenseNet-100-12 | 23.23±0.41 | 22.70±0.41 |

Table 4.8 – ResNet incremental training results for CIFAR-100. We start with a ResNet trained on 90 categories and then add the 10 left out categories.

| Train type | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|
| 200 + 200 | 31.95 | 31.74 | 32.61 | 28.64 | 32.21 |
| 200 + 100 | 31.95 | 32.30 | 33.24 | 27.57 | 32.67 |
| 200 + warmup(50) + 100 | 31.95 | 32.31 | 33.23 | 28.06 | 32.71 |
| 200 + warmup(20) + 100 | 31.95 | 32.28 | 33.24 | 27.42 | 32.65 |
| 200 + 200 Distillation | 31.95 | 30.20 | 31.59 | 28.93 | 31.33 |
| 200 + 100 Distillation | 31.95 | 30.72 | 32.16 | 27.91 | 31.73 |
| 200 + 100 Distillation 2nd half | 31.95 | 30.99 | 33.01 | 30.66 | 32.77 |
| (R) 200 + 200 | 30.83 | 31.47 | 33.04 | 26.93 | 32.43 |
| 200 + 200 Adam | 31.35 | 32.23 | 33.10 | 27.74 | 32.56 |

Table 4.9 – DenseNet incremental training results for CIFAR-100. We start with a DenseNet trained on 90 categories and then add the 10 left out categories

| Train type | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|
| 300 + 300 | 23.73 | 23.48 | 23.96 | 22.70 | 23.84 |
| 300 + 300 Distillation | 23.73 | 22.48 | 23.81 | 22.70 | 23.70 |
| 300 + 150 Distillation | 23.73 | 22.95 | 24.32 | 22.53 | 24.14 |

frozen. We start from the blocks closest to the input. The networks have 3 blocks, so we train only 1 block and 2 blocks, to see how the result differs from training all 3 blocks. Results are shown in Table 4.10. As expected they perform worse than training all the blocks. This implies that when new categories are added, the lower layers (layers close to the input) need to be adapted and are not general enough to provide good performance on all categories.

### 4.4.6 Effect of optimiser

The optimiser has a large effect on the final parameter values. Choosing the correct learning rate and weight decay values are critical for the training to arrive at good

Table 4.10 – Training different number of blocks of ResNet on CIFAR-100

| $\alpha$ | #Trained Blocks | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|---|
| 0.1 | 1 | 31.95 | 32.27 | 33.28 | 29.95 | 32.95 |
| 0.2 | 1 | 31.95 | 33.72 | 34.97 | 28.14 | 34.29 |
| 0.1 | 2 | 31.95 | 32.34 | 33.28 | 27.63 | 32.72 |
| 0.2 | 2 | 31.95 | 34.30 | 35.42 | 26.67 | 34.54 |

parameter values. In this section we briefly compare the training of a deep neural network using different optimisers and various setting for their hyper-parameter values.

Stochastic gradient descent (SGD) is arguably the most popular optimiser for training deep convolutional networks. Most architectures are trained with SGD and a step learning rate decay schedule. We have seen in our experiments in the previous section that such a schedule for the learning rate leads to sub-optimal results when training to extend a model with new tasks. In contrast to SGD, the Adam optimiser [51] performs weight updates as a function of gradients and a per-parameter learning rate. In the case of SGD, the learning rate for all parameters is the same. Having the ability to apply per-parameter learning rates seems to be a good idea because this will update each paramter differently. If parameters critical for older tasks are not changed too much from their existing values, it might be possible to achieve faster training and prevent drop in performance.

We first compare training on all categories of CIFAR-10 using SGD and Adam. Table 4.11 shows the result of training on all categories for one training cycle starting from random initialisation. The learning rate for SGD has been optimally found by the community. Since Adam is not commonly used, we experiment with different values and report the results corresponding to the best validation error rates. We see that there is a substantial gap in the error rate obtained by SGD and Adam (7.73% and 9.74%). This implies that Adam is not able to achieve the same level of performance as the optimally tuned settings for SGD.

Table 4.11 – Comparison of optimisers for CIFAR-10

| Architecture | Optimiser | 1 cycles | 1.5 cycles | 2 cycles |
|---|---|---|---|---|
| ResNet-20 | SGD (0.1) | 7.73±0.20 | 7.78±0.15 | 7.44±0.17 |
| ResNet-20 | Adam (0.01) | 13.08±0.24 | 13.31±0.20 | 12.98±0.24 |
| ResNet-20 | Adam (0.001) | 9.74±0.10 | 9.52±0.20 | 8.97±0.37 |
| DenseNet-100-12 | SGD (0.1) | 4.80±0.16 | 4.96±0.17 | 4.69±0.27 |

Our primary objective is however to evaluate the continual learning step where

new tasks are added to a given trained model. We compare the different optimisers at this step. In the case of SGD, a high learning rate is required to move out of the local minima obtained for the older tasks. However this leads to worsening of the performance on older tasks. Adam which has per-parameter adaptive learning rates can use the gradient magnitude to enable parameter values to adapt in a more convenient manner for old and new tasks. We show the results in Table 4.12. We see that the same pattern holds for Adam as it does for SGD. There is a drop in performance for the older categories and newer categories do not reach the best possible performance level. This leads to an overall worse error rate when compared to the training of all categories together.

Table 4.12 – Comparison of optimisers during task additions in CIFAR-10

| Optimiser | e-10-9-9-9 | e-10-10-9-9 | e-10-10-10-9 | e-10-10-10-1 | e-10-10-10-10 |
|---|---|---|---|---|---|
| 200 + 200 SGD (0.1) | 7.73 | 6.76 | 7.34 | 7.58 | 7.36 |
| 200 + 200 Adam (0.01) | 12.3 | 11.83 | 12.67 | 13.54 | 12.76 |
| 200 + 200 Adam (0.001) | 9.20 | 10.70 | 11.58 | 12.20 | 11.64 |

We see that the choice of the optimiser has an important effect on the final performance of a model. However, the task addition step shows similar behaviour for both SGD and Adam. One implication of this is that future continual learning algorithms need to explicitly take into account task specific and task agnostic parameters. Task specific parameters should not change drastically when the model is presented with newer tasks. However task agnostic parameters need to adapt to provide a general performance boost for all tasks involved.

### 4.4.7 Weight space analysis

Our experiments till now have shown that training on tasks incrementally does not perform as well as training on all the tasks simultaneously. The main difference between these trainings is the starting weights of our model. In the simultaneous training case we start with a randomly initialised model, while in the second phase of incremental training we start with weights that are optimised for the older tasks. We can characterise the difference between the models by measuring the *distance* between models in weight space. We compute the distance between corresponding parameters in the two models and report the mean over all the parameters.

We compute the $L_2$ distance between weights from different scenarios, before and after training has converged. The results are summarised in Table 4.13. The $L_2$ distance between tasks trained incrementally and random initial weights (inc_train, init) is 84.11. The distance between all_train, init is 90.69. We see that in the case of training incrementally the weights have a smaller distance

from the initialisation point but a large distance from the simultaneously trained weights. One hypothesis about why the performance of incremental training is inferior is that the the learned weights do not move away enough from the starting point. The average $L_2$ distance between training on all tasks simultaneously, and training the same tasks incrementally (all_train, inc_train)is 105.42. In contrast the average distance between training runs is $\sim 110$.

The high distance between different training runs of the same models may be expected since it is known that in deep neural networks there are multiple equivalent local minima present [50], and it is not clear wether they will be close together or far apart in parameter space.

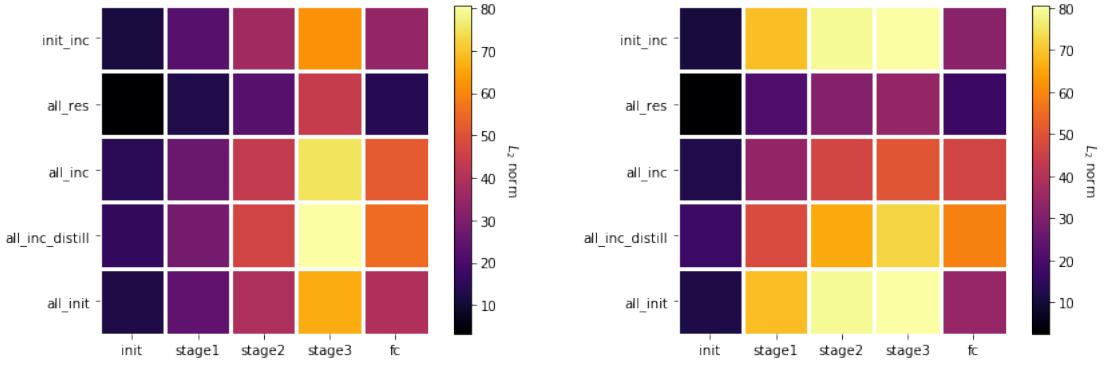
The changes in weight can be measured at a finer scale by looking at the $L_2$ norm of weight distance per stage in a network. In convolutional networks such as ResNet and DenseNet, the layers are divided into several stages, each stage operating at a fixed spatial resolution. Figure 4.7 shows the change is weight distance per stage between several weight pairs corresponding to different training setups.

- First we note that comparing between random initialisation and complete training of all categories (`all_init`), the deeper layers have moved more from the initialisation as compared to layers closer to the input.
- In the case of `all_res`, since we are doing a double training with restart, the weights do not change much. The majority of the change happens in `stage3` which means that there is an improvement in the more abstract high level features.
- The weight distance in case of `all_inc` is maximum in `stage3` which implies that the high level features differ by a large amount. Both of the compared models are trained on the same data and classify over the same categories. The weight distance does not tell us which features are better directly. However, since the classification performance is better when training on all categories, one conclusion is that the features are better when training on all categories simultaneously. A support for this fact is that the weights for `fc` layer change comparatively less.
- Incremental training with distillation shows a large weight distance norm for `stage3` weights as compared ordinary incremental training.
- The observations for DenseNet training are similar to that of ResNet.

In order to explore the weight space and characterise it, we can interpolate between the considered models in weight space. To do this, we choose two models

Table 4.13 – CIFAR-100 training with ResNet, $L_2$ distance between models in weight space.

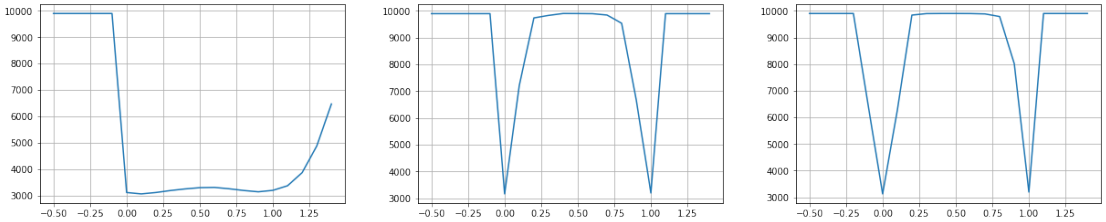
| Weight pairs | $L_2$ |
|---|---|
| all train, all train restart | 53.60±0.09 |
| all train, inc train | 105.42±1.46 |
| inc train, init | 84.11±1.94 |
| all train, init | 90.69±0.25 |
| all train restart, init | 91.23±0.11 |
| init, other init | 56.57±0.10 |
| all train, other init | 90.80±0.12 |
| mean distance between runs | ∼110 |
| mean distance between inc runs | ∼100 |




(a) (b)

Figure 4.7 – Change in weights per stage, measured as $L_2$ norm, for ResNet-20 (Left) and DenseNet-100-12 (Right), on CIFAR-100.

and construct a new model by interpolating the weight values between the considered models. We then compute the corresponding validation errors for the model having these interpolated weight values (see Figure 4.8). We see that between the models trained on all data, one training cycle and two training cycles with restart, there is quite a low barrier. In contrast, between the model trained on all data and the model trained incrementally, we see a high cost barrier. This is one potential explanation why the validation error is much worse when trained incrementally in two steps, as compared to training on all data at once. The starting point of the incremental step, that is the trained model, is a worse point in weight space as compared to random initialisation.




(a) (b) (c)

Figure 4.8 – ResNet validation error as a function of interpolation between weight vectors. (a) interpolation between ResNet trained for 200 epochs and ResNet trained on 200 + 200 epochs with restart. (b) ResNet trained for 200 epochs and ResNet trained incrementally over two steps. We see that the figure on the left shows an easy path between the two points. However, in the incremental case there is a clear obstruction. (c) ResNet trained incrementally with distillation loss for older categories.

## 4.5 Discussion on related work

Generative Adversarial Networks (GAN) [29] were introduced in § 2.5. Here we briefly highlight the important points again. GANs are a type of generative model whose training procedure involves a two-player game between a generator network and a discriminator network. The generator network transforms a random noise input into a data sample. The discriminator network classifies samples as a real sample or a generated sample. The discriminator network serves as the loss function for training the generator network, as the objective of the generator network is to produce samples which the discriminator classifies as real. There has been several improvements to the original GAN model involving architectural and optimisation advancements. The recently proposed BigGAN [4] model presents a framework for training GANs and produces very visually rich samples that are quite hard to distinguish from real data.

Conditional-GAN [70] presents the framework for sampling from a GAN based on a conditioning factor, such as image category. Most early work on GAN had focused on unconditional training of GAN. There is no clear evidence as to where in the network the conditioning information is to be provided and different techniques have been proposed, such as providing a one-hot class embedding at the input, or a learnt class embedding in one of the internal layers. Conditional batch normalisation [12] layers have been found to be effective for modulating the activations in a network and have been subsequently used for conditional image generation in GANs [4, 72]. GANs were chosen as the generative model of choice because out of the current generative models, these have produced samples which arguably

are the most visually accurate. Hence this motivates us to study them from a classification perspective, and ask the question if these visually sound images can be used for training classifiers.

**View of GANs from a classification perspective.** Training and evaluating classifiers on GAN samples have been investigated in [94], where the authors have compared the classification error with other evaluation metrics such as Inception Score (IS) and Fréchet Inception Distance (FID). Their aim is to use the classification error on train and test data sets as an evaluation metric and they are not concerned with generating samples that lead to training of good classifiers. In this work our aim is not to compare different metrics and improve them but to solely improve the validation error of a classifier on real test samples, after training a classifier on generated samples only. From this perspective, our objective is really not to produce samples which are visually sound but to produce samples from which good discriminative features can be learnt.

A closely related work is that of [91]. This work focuses on a similar study as ours with recent state-of-the-art GAN models. However, there are several differences: i) [91] trains a separate GAN model per category, whereas we train class conditional GANs. Conditional GANs should result in better GAN models since the learning procedure has access to more data and may also help the model to learn to generate more diverse samples by making use of inter class information. Training separate GANs per category is also not a scalable strategy as the number of categories in the data set increases. ii) We aim to see how the samples from a BigGAN model, which are visually meaningful, compares to real data samples in the context of training a good classifier. iii) We propose methods to obtain an improvement in training classifier from generated samples only.

**Replay buffer** The idea of using a replay buffer has been part of rehearsal based techniques to prevent catastrophic forgetting [83]. However these early experiments were done on smaller scale neural networks. It is not obvious how these findings carry over to modern over parameterised deep neural networks. In this work we use replay buffers that contain samples both from real data sets and from a generative model. In the context of continual learning, without having access to real data samples, one approach is to construct a pseudo replay buffer using a generative model of the training set [93]. A generative model is learnt on older data sets, and samples from this generative model are combined with samples from the current data set. This prevents forgetting as the samples from the generative model are used to learn parameters that are optimised for old and new tasks. In practice these approaches are limited by the quality of the generative model. We analyse the situation by considering BigGAN, the current state-of-the-art GAN

model, in terms of sample diversity and sample quality. The data sets we consider comprise more complex natural images as compared to digit data sets considered in [93].

We further experiment with a replay buffer considering real data samples. This serves two purposes: first, it is not clear how the training will proceed if we remove the restriction of not having access to older task data. So using old task data we can do an analysis on various training strategies. Second, using real data simulates having access to a *perfect* generative model which can produce samples as good as from the real data set. To the best of our knowledge, we did not find any related work for this scenario of training with real data samples, hence we believe it is a novel contribution and analysis.

## 4.6 Conclusion

**Training classifiers from a GAN** We have investigated the samples from a state of the art generative model, BigGAN [4], from the perspective of training a classifier. We saw that the samples generated from this model look quite realistic. However if we want to use these samples for a downstream task, such as training a classifier, there is still a gap in the performance between generated samples and real data samples. A possible reason for this gap is that the GAN model is not able to completely capture the complexity of the underlying data distribution. On the other hand, from a compression point of view, a generative model in this case is more efficient for storage as compared to storing the same number of real data samples that obtain similar classification error. In order to improve the classification performance we used a pre-trained classifier to filter out samples based on the predicted category and prediction probability. In our experiments we see that using a threshold for the prediction probability is essential in cases where there are a large number of categories involved. To help the generative model to cover more of the underlying real data distribution, we used multiple GAN sampling, leading to significant reduction in the gap of classifier error between training on real data and GAN samples.

We notice that currently used metrics to measure the quality of a generative model, such as IS and FID are not completely indicative of the classifier generalisation error. We also saw that some of the techniques led to small or negligible improvements in these metrics but significant improvement in classification error.

**Analysis of continual learning with replay buffers** We see in our experiments that it is not straightforward to add even one new category to a model that can classify 9 existing categories, without extensive re-training. The same holds for adding multiple categories. Our objective is to add the new categories

without going through the entire training process. However we see that in the methods that we tried, shorter training periods compromise the performance on the combined categories. This indicates that accommodating one new category requires significant re-adjustment of the existing weights. A weight space analysis seems to indicate a high loss barrier between solutions obtained by training on all categories at once, and training incrementally on the categories.

One way to reduce the re-training time when adding new categories, is to train for fewer epochs. However, training with a low learning rate prevents the new categories from converging to optimum weights. In contrast, a higher learning rate is required to initially escape the local mimima that the weights are in for the old categories, so as to accommodate the newer categories. Resuming training with a high learning rate in turn leads to requiring more epochs to find the optimum for all categories combined. The choice of the optimiser and the associated learning rate decay schedule plays a big part in this training procedure. It is possible that we need a different optimisation procedure, or a different manner of adjusting the learning rate, for finding an optimum set of weights for all tasks combined.

On a more positive side, by preserving the classifier responses corresponding to the older categories when we train on the new categories, we are able to improve on the performance of the entire model. We achieve this by using a distillation loss [41] between the responses of the old categories in the new and old model. It is possible that other such criteria can be defined to better accommodate old and new tasks in one model.

# Chapter 5

# Coupled Ensembles

Convolutional networks are one of the widely used architectures in deep learning. They are the model of choice in most image related tasks, ranging from image classification to video analysis. The `convolution` operation leads to an efficient parameterisation of the learnable weights in the network and also introduces spatial invariance as a property that is part of the network. In this chapter we introduce, *coupled ensembles* as an architecture of convolutional networks which leads to reduced number of parameters without loss in generalisation error. Coupled ensembles also lead to faster training times and lend themselves to an efficient model parallel training scheme by design.

The inspiration for coupled ensembles arose from our semantic hierarchy predictions in Chapter 3. We saw that we had gains in performance when the CNN architecture had parallel branches as compared to not having any parallel branches or when they were separately trained. However we also had a probability adjustment step which contributed to the overall gain in performance. Thus, the idea of coupled ensembles arose from analysing the effect of gain in performance coming from having an architecture with parallel computation paths. CNNs are used for feature learning in a variety of tasks, which means having more efficient CNN architectures will benefit all such downstream tasks.

## 5.1 Design of deep convolutional networks

The architecture of deep convolutional networks includes several design factors, namely kernel size, number of kernels, number of layers. The convolution operation has parameters which control the size of its output which include stride and padding. Convolutional networks started with having a non uniform design. LeNet [59] and AlexNet [54] have different kernel sizes at each layer. The layers closest to the input have larger kernel sizes while deeper layers have smaller kernel

sizes. The reasoning is that the shallower layers with larger kernels extract coarse features while the deeper layers learn finer features over a small receptive field. The number of output feature maps at each layer is different, with deeper layers creating more output feature maps. Again, we can reason that deeper layers steadily learn a larger number of features to make the classifier more discriminative. Along this computational pipeline of a sequence of layers there are some downsampling layers. Downsampling layers help to aggregate features across the spatial dimension. Additionally downsampling reduces the spatial dimension which prevents memory explosion. This is needed as the deeper layers tend to have a larger number of output feature maps.

We can think that one of the first steps towards having a design template for convolutional networks was with the introduction of VGGNet [96]. VVGNet started the practice of using the same kernel size in all layers, keeping the number of kernels constant as long as the spatial dimension of the input did not change, and doubling the number of kernels following a downsampling operation. The first layer of AlexNet had a convolution layer with kernel size of $7 \times 7$ . VGGNet replaced this with 3 layers, each with kernel size of $3 \times 3$. The effective receptive field is still the same but instead of computing features directly over the input image, now it is being done in a learnt feature space. Moreover VGGNet adopted the use of the same kernel size of $3 \times 3$ at all layers. The convolution operation was carried out with padding such that the spatial dimension does not change after applying the convolution. The spatial dimension was changed only at a few layers, by using either `maxpool` or strided convolutions [97]. Each set of layers which operate at a particular spatial dimension is called a "stage". We see that going from the shallower to deeper layers, after each downsampling stage, the spatial dimension is halved and the number of kernels (and hence the output feature maps) is doubled. This means that the total number of parameters is roughly the same in each stage. In summary with the introduction of VGGNet, we have the usage of $3 \times 3$ kernels, same number of output feature maps inside each stage, downsampling only at specific points, and doubling the output feature maps following the downsampling.

In VGGNet there are the final two layers which are `fc` layers. By nature of their dense connectivity these layers contain a large portion of the total learnable parameters in the entire network, even though they contain the minority of the computations. Network in Network [62] introduced the idea of removing the `fc` layers and replacing them with convolution layers which output feature maps having dimension of $C \times 1 \times 1$. This is equivalent to the $C$ outputs which we get from a `fc` layer. Similar ideas of a all convolutional network have been presented in [97] and ResNet [35]. Finally, in addition to these, Residual Networks [35] proposed the idea of using identity skip connections among non-continuous layers.

These elements form the template using which most convolutional architectures are designed in the context of classification. Highway Networks [99] also use skip connections but the learnt gating mechanism performs worse in practice as compared to the identity skip connection of the ResNet. DenseNet [45] replaces the addition operation in ResNet skip connections with concatenation.

## 5.2 Coupled ensembles

In this section we introduce "coupled ensembles", the required terminology and explain the working of the architecture in detail. Coupled ensembles is an extension of the template described in the previous section. The idea is to factorise the parameters into multiple parallel computation paths, where each path receives the same input and produces a feature representation. These representations are combined to make the final prediction. This design achieves generalisation error comparable to state of the art models with a significantly *lower* parameter count. Figure 5.2 shows the train and test version of coupled ensembles. The implementation is straightforward and can be combined with the design elements we discussed. Our implementation to compose different standard architectures is available at: https://github.com/vabh/coupled_ensembles.

**Terminology.** We define some terms for the following discussions:

- **Branch:** Each architecture comprises one or more branches. The number of branches is denoted by $e$. Each branch takes as input a data sample and produces a score vector corresponding to the target classes. Current design of CNNs are with a single computation path, and are referred to as single-branch ($e = 1$).

- **Element block:** This defines the architecture used to form a branch. In our experiments, we use DenseNet-BC [45] and ResNet [37] with pre-activation as element blocks.

- **Fuse Layer:** The operation used to combine the parallel branches which make up our global model. In our experiments, branches are combined by taking the average of score vectors, e.g. log probabilities for target classes. The fuse layer can be used at any point in the computation path where a score vector is available. In § 5.3.3 we explore different choices for the level of fuse layer during training and prediction.

In this chapter, all discussion is in the context of an image classification task where each image sample belongs to exactly one class out of a set of pre-defined

classes. This is the case for CIFAR [53], SVHN [74] and ILSVRC [87] datasets. In theory, this generalises to other tasks as well such as image segmentation, object detection, etc.

We consider neural network models which output a score vector of the same dimension as the number of target classes. This is usually implemented as a linear layer and referred to as a fully connected (`fc`) layer. This layer can be followed by a `softmax (sm)` layer to produce a probability distribution over the target classes as illustrated in Figure 5.1 top. During training, this is followed by a loss layer, for example, negative log-likelihood (nll) as shown in Figure 5.1 bottom. This is the case for most architectures used for image classification [54, 96, 100, 37, 107, 45]. We note that all that is required is that the architecture produces a score vector corresponding to the number of categories, and this can be produced by operations other than a `fc` layer, such as `conv1x1`. The coupled ensemble approach may be adapted to multi-label classification by replacing the `sm nll`) layers by corresponding layers that are appropriate for multi-label classification.

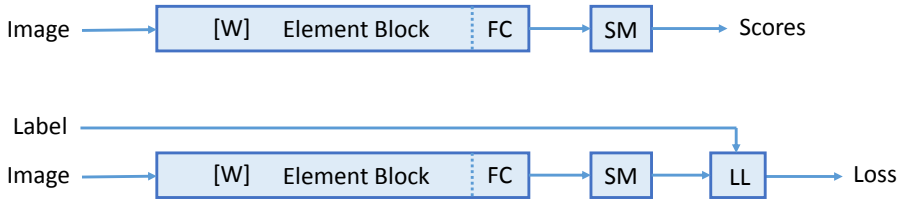


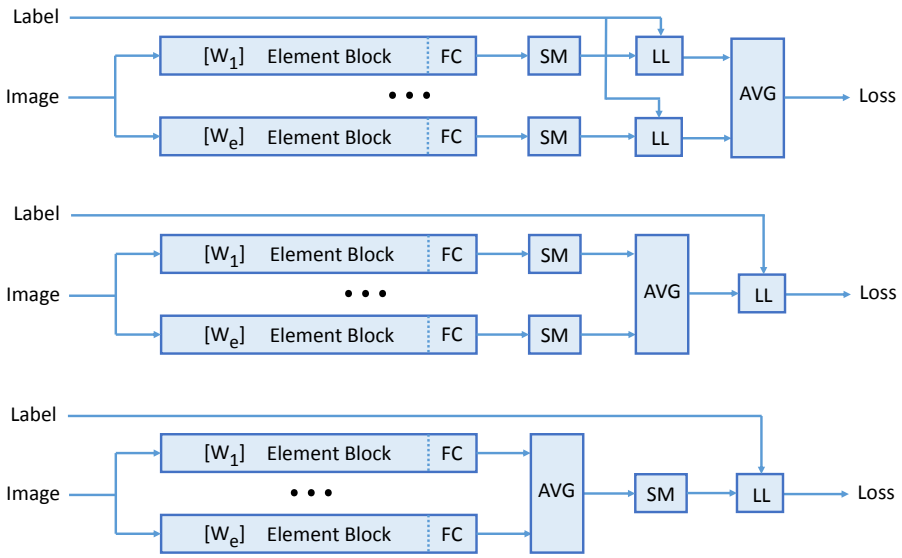

Figure 5.1 – Versions of the element network. Top: test, bottom: train.

In summary, the coupled ensemble approach requires only a score vector produced by underlying element blocks. It is agnostic to the internal architecture of the element block, however complex it may or may not be. Each element block takes the same input and outputs a score vector which is combined through the fuse layer in the forward pass of training. The loss is computed after the fuse layer operation. Gradients are computed and backpropagated back to all the element blocks. The transformation of having multiple branches combined through the fuse layer during training leads to a performance improvement with a *lower* parameter count in all our experiments (see § 5.3). The parameter vector $W$ of the composite branched model is the concatenation of the parameter vectors $W_e$ of the $e$ element blocks with $1 \leq i \leq e$. There is no increase in the number of parameters as all parameters are in the "element blocks" and the "fuse layer" does not contain any parameters.

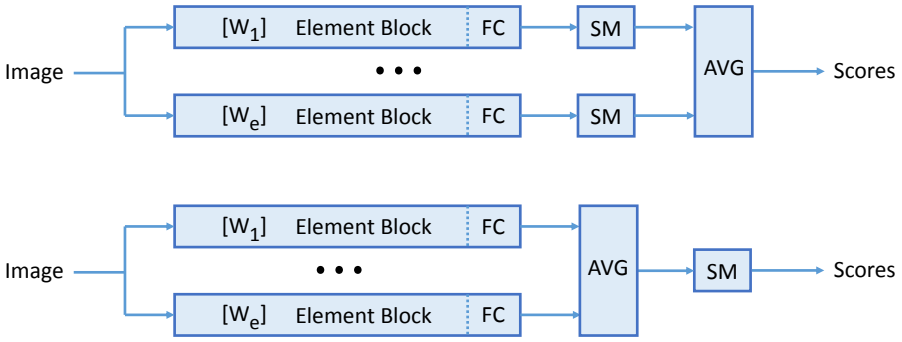
**Independent ensembles** In the case of ensemble of *independently trained* models, first each individual prediction is obtained and then combined using some logic

to make the final predictions. The important point to note is that each of the $e$ instances are trained *separately*.

In the case of coupled ensembles, the single model is composed of parallel branches and each branch produces a score vector for the target categories. The score vectors are combined by the "fuse layer" *during training* and the model produces a single prediction. The gradient of the loss received in each branch is the same, and it depends on the predictions made by *all the other branches* in the corresponding forward pass. Moreover, there is a global budget on the number of parameters and we compare the efficiency of coupled ensembles with other models with similar number of parameters contained in a single branch.




(a) Train version. Top: LL fusion, middle: LSM fusion, bottom: FC fusion.




(b) Test version Top: SM (classical) fusion, bottom: FC fusion. AVG: averaging layer.

Figure 5.2 – Train and test setup of coupled ensembles.

**Fuse layer operation.** During training we can insert the fuse layer at one

of three places (Figure 5.2a) and one of two places during testing (Figure 5.2b). These choices are:

- Activation (FC) average: Average the output of the `fc` layer of each branch.
- Probability (LSM) average: Average the log-probabilities for each branch. This can be obtained by applying the `log_softmax` after the `fc` layer. Note that, for inference, averaging the `fc` layer activations is equivalent to averaging the log-probabilities (see § 5.5.1).
- Log Likelihood (LL) average: Average the loss of each branch. This option should be the same as training each element block independently.

We explore combinations of these choices and discuss in more detain in § 5.3.3. Further implementation details are given in § 5.5.

## 5.3 Experiments

### 5.3.1 Data sets

We use the CIFAR [53] and SVHN [74] data sets for validating the proposed architecture. CIFAR-10 and CIFAR-100 consist of 50 000 training images and 10 000 test images, belonging to 10 and 100 categories respectively. SVHN consists of 73 257 training images, 531 131 "easy" training images (we use both for training) and 26 032 testing images belonging to 10 categories. All images have a size of 32×32 pixels.

### 5.3.2 Model training

All hyper parameters and data augmentation during training are the same as in the original descriptions of the "element block" given in their corresponding papers. This may not be the optimal setting in our case (especially the learning rate decay schedule) but we do not alter them, so as to not introduce any bias in comparisons. For CIFAR-10, CIFAR-100 and SVHN, the input image is normalised by subtracting by the mean image and dividing by the standard deviation. During training on CIFAR data sets, standard data augmentation is used, which comprises random horizontal flips and random crops. For SVHN, no data augmentation is used. However, a dropout ratio of 0.2 is applied in the case of DenseNet when training on SVHN. Testing is done after normalising the input in the same way as during training. All error rates are given in percentages and correspond to an average of the last 10 epochs. This measure is more conservative than the one used

by the DenseNet [45] authors. All execution times were measured using a single NVIDIA 1080Ti GPU with the optimal micro-batch (see § 5.5.2).

Experiments in § 5.3.3 and § 5.3.4 are done on the CIFAR-100 data set with the "element block" being DenseNet-BC [45], depth $L = 100$, growth rate $k = 12$. For experiments in § 5.3.5, we consider this same configuration (with a single branch ($e = 1$)) as our baseline reference point.

### 5.3.3 Fuse Layer choice

In this section we compare the different fuse layer choices during training and testing. The results are shown in each half of Table 5.1 for two different number of element blocks ($e = 4$ and $e = 2$).

The first three rows in each half of Table 5.1 correspond to the three possible choices for the fuse layer during *training* (see Figure 5.2a) and they are marked by the "Fuse layer" column. The fourth row shows results from training each element block separately, which is equivalent to removing the fuse layer. This is also the same case as training with the "LL" fuse layer.

The `FC` and `SM` columns correspond to the fuse layer operation during testing (see Figure 5.2b). The `Indiv.` column shows the error rate when each element block is evaluated separately, regardless of whether they were trained jointly or independently.

We can make the following observations based on results from the top half of Table 5.1:

- The average error rate of each of the "element blocks" trained jointly in coupled ensembles with LSM fuse layer (second row) is significantly lower than the error rate of the individual instances trained separately (fourth row). This indicates that the coupling not only forces them to learn complementary features as a group but also to learn better representations, individually. Averaging the log probabilities forces the network to continuously update all branches so as to be consistent with each other. The error gradient that is back-propagated from the fuse layer is the same for all branches, and this gradient depends on the *combined* predictions. This means that at every step all branches act complementary to the other branches' weight updates.
- When training with `FC` fuse layer, the individual branches do not perform well. This is expected since a similar FC average may be reached with quite unrelated outputs. Depending on the initial weights, the output of each element block can be vastly different during the early stages of trainng. The `FC` training with `SM` prediction works a bit better but is still inferior because the non-linearity of the `sm` layer distorts the FC average. `FC` training with `FC` prediction works quite well though it does not yield the best performance.

| L | k | e | Fuse layer | `Indiv.` | `FC` | `SM` | Params. | Epoch | Test |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $e = 4$ | | | | | |
| 100 | 12 | 4 | FC | 74.36±26.28 | 22.55 | 31.92 | 3.20M | 402 | 2.00 |
| 100 | 12 | 4 | LSM | 22.29±0.11 | **17.61** | 17.68 | 3.20M | 402 | 2.00 |
| 100 | 12 | 4 | LL | 22.83±0.18 | 18.21 | 18.92 | 3.20M | 402 | 2.00 |
| 100 | 12 | 4 | none | 23.13±0.09 | 18.42 | 18.85 | 3.20M | 341 | 2.00 |
| | | | | $e = 2$ | | | | | |
| 100 | 12 | 2 | FC | 52.68±22.95 | 22.25 | 28.78 | 1.60M | 174 | 0.98 |
| 100 | 12 | 2 | LSM | 22.17±0.32 | **19.06** | 19.43 | 1.60M | 174 | 0.98 |
| 100 | 12 | 2 | LL | 22.78±0.08 | 19.33 | 19.91 | 1.60M | 174 | 0.98 |
| 100 | 12 | 2 | none | 23.13±0.15 | 20.44 | 20.44 | 1.60M | 171 | 0.98 |

Table 5.1 – Coupled ensembles of DenseNet-BC ($e = 4$, $e = 2$) with different "fuse layer" combinations. Columns "$L$" and "$k$" define the "element block" architecture, column "$e$" is the number of branches. Column "Fuse layer" indicates the type of "fuse layer" during training (see § 5.2); "none" for separate trainings. Column `Indiv.` is the mean error rate of each branch; Columns `FC` and `SM` give the error rate for "fuse layer" choices during testing. Column "Epoch" is time taken to complete one training epoch, "Test" is the testing time per image. The mean and standard deviation of all error rates are computed from 4 runs.

- The `FC` prediction works at least as well and often significantly better than the `SM` prediction. This can be explained by the fact that the `sm` layer normalises values to probabilities (between 0 and 1), while the FC values remain spread out over a greater range (have lower entropy) and preserve more information from the individual element blocks during the final prediction.
- The results of the third row, `LL` fuse layer, are similar to the fourth row, which is individual training. This is expected and serves as a 'unit test' for our implementation. The difference between the third and fourth row training manifests in the implementation. In the third row we train 4 element blocks, for which we combine the final loss values. Hence each element block trains the same as if they were being trained separately. This is in contrast to the other fuse layer operations where the intermediate outputs (`fc` or `(l)sm`) are combined before computing the final loss.
- The bottom half of Table 5.1 shows results for experiments with two branches ($e = 2$). The observations are the same as with four branches. Even using only two branches provide a significant gain over a single-branch architecture

of comparable size.

These experiments show that coupled ensemble training leads to improvement over separately trained models, when the final predictions are combined. The constituent element blocks each benefit from coupled ensemble training which lead to better generalisation when the individual predictions are combined. For the choice of fuse layer operation, `LSM` performs best and we use this for both training and testing for the rest of the experiments.

### 5.3.4 Comparison between single branch and coupled ensemble models

In the previous section we compared coupled ensembles with different fuse layers and found that `LSM` fuse layer obtained the best generalisation. We now compare coupled ensembles with `LSM` fuse layer to single branch ($e = 1$) models having comparable number of parameters. The two halves of Table 5.2 show the error rates of single-branch DenseNet models with different width (growth rate) $k$ and depth $L$ configurations. The choice of single-branch models has been done by: increasing $k$ while keeping $L$ constant, by increasing both $k$ and $L$, or by increasing $L$ while keeping $k$ constant. We can make the following observations:

- The error rate of the various single branch models are quite stable.
- Coupled ensemble training does significantly better than a single branch model. The error rate of the best single branch model is 20.01%, with about 3.2M parameters. Keeping the same parameter budget, using 4 branches reduces the error rate to 17.61% ($-2.40$) for the coupled ensemble model. An ensemble of the independently trained models obtain 18.42% ($-1.59$), which is worse than the result of the corresponding coupled ensemble model but better than the single branch model.
- The coupled ensemble model with $e = 4$ and `LSM` "fuse layer" has almost the same performance as a DenseNet-BC ($L = 250, k = 24$) model [45] (17.61 versus 17.60), which has about *5 times more* parameters (15.3M versus 3.2M).

These observations show that arranging a given budget of parameters into parallel branches is more efficient in terms of parameters, as compared to having a large single branch model or multiple independent models. As we observed in this section, it is possible to reduce the number of parameters to a great extent. In § 5.3.5, we analyse the relation between the number of branches and the model performance.

Table 5.2 – Coupled ensembles of DenseNet-BCs compared to single branch models; see legend of Table 5.1.

| L | k | e | Fuse layer | `Indiv.` | `FC` | `SM` | Params. | Epoch | Test |
|---|---|---|---|---|---|---|---|---|---|
| | | | | $e = 4$ | | | | | |
| 100 | 12 | 4 | LSM | 22.29±0.11 | **17.61** | 17.68 | 3.20M | 402 | 2.00 |
| 100 | 12 | 4 | none | 23.13±0.09 | 18.42 | 18.85 | 3.20M | 341 | 2.00 |
| 100 | 25 | 1 | n/a | 20.61±0.01 | n/a | n/a | 3.34M | 164 | 0.8 |
| 154 | 17 | 1 | n/a | 20.02±0.10 | n/a | n/a | 3.29M | 245 | 1.3 |
| 220 | 12 | 1 | n/a | 20.01±0.12 | n/a | n/a | 3.15M | 326 | 1.5 |
| | | | | $e = 2$ | | | | | |
| 100 | 12 | 2 | LSM | 22.17±0.32 | **19.06** | 19.43 | 1.60M | 174 | 0.98 |
| 100 | 12 | 2 | none | 23.13±0.15 | 20.44 | 20.44 | 1.60M | 171 | 0.98 |
| 100 | 17 | 1 | n/a | 21.22±0.12 | n/a | n/a | 1.57M | 121 | 0.67 |
| 124 | 14 | 1 | n/a | 21.75±0.10 | n/a | n/a | 1.55M | 135 | 0.77 |
| 148 | 12 | 1 | n/a | 20.80±0.06 | n/a | n/a | 1.56M | 159 | 0.90 |

### 5.3.5 Choice of the number of branches

In the previous two sections, we saw that coupled ensembles with 2 and 4 branches led to improvement in performance over single branch models, for a fixed parameter budget. In this section, we investigate the optimal number of branches $e$ for a given model parameter budget. We evaluate on CIFAR-100, use DenseNet-BC [45] as the "element block", and parameter budget equal to $0.8M$ (number of parameters in DenseNet-BC ($L = 100, k = 12$)). The optimal number of element blocks is likely to depend upon the network architecture, upon the parameter budget and upon the data set but this gives at least one point of reference. Related results for larger models are in Table 5.4 (last four rows).

In Table 5.3 we show the error rate for different configurations of branches $e$, depth $L$, and growth rate $k$. One difficulty in choosing the different configurations of DenseNet-BC architecture for a target parameter count, is that there are constraints in choosing the $L$ and $k$ values ($L$ has to be a multiple of 6 modulo 4). This is quite critical in moderately sized models, like the 800K parameter model targeted here because each element block itself needs to have some basic representation power in order to contribute to the composite coupled ensemble. We selected model configurations with about the same parameters to have a fair comparison. A few models have slightly more parameters so that some interpolation can be done for possibly more accurate comparisons. We can make the following

Table 5.3 – Different number of branches, $e$, for a fixed parameter budget. The models are trained on CIFAR-100 with standard data augmentation. See caption of Table 5.1 for the meaning of row and column labels. For $e > 1$, "fuse layer" is `LSM`. (*) Average and standard deviation on 10 runs with different seeds.

| L | k | e | `Indiv.` | `FC` | `SM` | Params. | Epoch | Test |
|---|---|---|---|---|---|---|---|---|
| 100 | 12 | 1 | **22.87±0.17(*)** | n/a | n/a | 800k | 86 | 0.51 |
| 76 | 10 | 2 | 25.58±0.20 | 21.66 | 22.17 | 720k | 103 | 0.63 |
| 88 | 9 | 2 | 25.15±0.31 | 21.87 | 22.19 | 747k | 119 | 0.71 |
| 94 | 8 | 2 | 25.72±0.20 | 21.95 | 22.22 | 666k | 115 | 0.69 |
| 100 | 8 | 2 | 25.42±0.20 | 21.87 | 22.07 | 737k | 126 | 0.75 |
| 70 | 9 | 3 | 26.67±0.40 | **21.10** | 21.24 | 773k | 129 | 0.77 |
| 82 | 8 | 3 | 26.47±0.17 | **21.25** | 21.46 | 800k | 141 | 0.85 |
| 88 | 7 | 3 | 26.92±0.41 | 22.09 | 22.49 | 698k | 148 | 0.92 |
| 94 | 7 | 3 | 26.50±0.12 | 21.95 | 22.35 | 775k | 160 | 0.98 |
| 64 | 8 | 4 | 28.58±0.59 | 22.44 | 22.58 | 719k | 142 | 0.88 |
| 70 | 8 | 4 | 27.65±0.48 | 21.50 | 22.12 | 828k | 156 | 0.94 |
| 58 | 7 | 6 | 30.11±0.53 | 23.87 | 24.22 | 718k | 179 | 1.08 |
| 64 | 7 | 6 | 30.65±0.62 | 23.08 | 23.36 | 840k | 198 | 1.20 |
| 58 | 6 | 8 | 32.15±0.00 | 25.95 | 25.70 | 722k | 219 | 1.35 |
| 64 | 6 | 8 | 31.52±0.38 | 24.42 | 24.69 | 843k | 250 | 1.51 |

observations:

- In the considered case (DenseNet-BC, CIFAR-100 and 800K parameters), the optimal number of branches is $e = 3, L = 70, k = 9$. With this configuration, the error rates decreases from 22.87 of the single-branch ($L = 100, k = 12$) DenseNet-BC model to 21.10 ($-1.77$).
- Using 2 to 4 branches yields a significant performance gain over the single branch, $e = 1$, case, and even over the original performance of 22.27% reported for the ($L = 100, k = 12$) DenseNet-BC in its paper.
- Using 6 or 8 branches performs significantly worse, possibly because the element blocks do not have sufficient capacity to learn meaningful representations.
- Model performance is robust to slight variations of $L$, $k$ and $e$ around their optimal values, showing that the coupled ensemble approach and the DenseNet-BC architecture are quite robust relative to these choices.

- The gain in performance comes at the expense of an increased training and prediction times even though the model size does not change. This is due to the use of smaller values of $k$ that reduces the throughput for smaller models.
- The variation of the depth $L$ and the growth rate $k$ are also evaluated for an approximately fixed parameter count. The performance is quite stable against variations of $(L, k)$ values.

The same experiment was done on a validation set, comprsing a 40K/10K random split of the CIFAR-100 training set and we could draw the same conclusions from there; they led to predict that the $(L = 82, k = 8, e = 3)$ combination should be the best one on the test set. The $(L = 70, k = 9, e = 3)$ combination appeared to be slightly better here but the difference is probably not statistically significant.

### 5.3.6 Comparison with state of the art

We next compare coupled ensembles to existing models of various sizes. We again use DenseNet-BC [45] architecture as the "element block" since this was the current state of the art at the time when we started these experiments. We also use ResNet [37] as an element block to evaluate the architecture agnostic property of coupled ensembles. Table 5.4 shows the current state of the art models (see § 5.6 for references) in the upper half and the performance of coupled ensembles in the lower half. All results presented in this table correspond to the predictions of single model. A further level of ensembling involving multiple models is considered in § 5.3.7.

Coupled ensembles with ResNet `pre-act` as element block and $e = 2, 4$ leads to a significantly better performance than single-branch models, which have comparable or *higher* number of parameters.

For the DenseNet-BC architecture, we considered 6 different model sizes, ranging from 0.8M up to 25.6M parameters. [45] reports results for the two extreme cases. We chose these corresponding values for the depth $L$ and growth rate $k$, and interpolated between them according to a log scale as much as possible. Our experiments show that the trade-off between $L$ and $k$ is not critical for a given parameter budget. This was also the case for choosing between the number of branches $e$, depth $L$ and growth rate, $k$ for a fixed parameter budget as long as $e \geq 3$ (or even $e \geq 2$ for small networks). For the 6 configurations, we experimented with both the single branch and coupled ensemble $(e = 4)$ versions of the model. Additionally, for the largest model, we tried $e = 3, 6, 8$ branches.

For single-branch DenseNet-BC, we obtained error rates higher than reported by [45]. From what we have checked, their Torch7 implementation and our PyTorch one are equivalent. The difference may be due to the fact that we used a

| Architecture | C10+ | C100+ | SVHN | #Params |
|---|---|---|---|---|
| ResNet $L = 110$ $k = 64$ [35] | 6.61 | - | - | 1.7M |
| ResNet stochastic depth $L = 110$ $k = 64$ | 5.25 | 24.98 | - | 1.7M |
| ResNet stochastic depth $L = 1202$ $k = 64$ | 4.91 | - | - | 10.2M |
| ResNet pre-act. $L = 164$ $k = 64$ [37] | 5.46 | 24.33 | - | 1.7M |
| ResNet pre-act. $L = 1001$ $k = 64$ | 4.92 | 22.71 | - | 10.2M |
| DenseNet $L = 100$ $k = 24$ [45] | 3.74 | 19.25 | 1.59 | 27.2M |
| DenseNet-BC $L = 100$ $k = 12$ [45] | 4.51 | 22.27 | 1.76 | 0.80M |
| DenseNet-BC $L = 250$ $k = 24$ | 3.62 | 17.60 | - | 15.3M |
| DenseNet-BC $L = 190$ $k = 40$ | 3.46 | 17.18 | - | 25.6M |
| Shake-Shake C10 Model S-S-I [24] | 2.86 | - | - | 26.2M |
| Shake-Shake C100 Model S-E-I | - | 15.85 | - | 34.4M |
| Snapshot Ensemble DenseNet-40 ($\alpha_0 = 0.1$) | 4.99 | 23.34 | 1.64 | 6.0M |
| Snapshot Ensemble DenseNet-40 ($\alpha_0 = 0.2$) | 4.84 | 21.93 | 1.73 | 6.0M |
| Snapshot Ensemble DenseNet-100 ($\alpha_0 = 0.2$) | 3.44 | 17.41 | - | 163M |
| SGDR WRN-28-10 [66] | 4.03 | 19.57 | - | 36.5M |
| SGDR WRN-28-10 3 snapshots | 3.51 | 17.75 | - | 110M |
| ResNeXt-29, 8×64d [107] | 3.65 | 17.77 | - | 34.4M |
| ResNeXt-29, 16×64d | 3.58 | 17.31 | - | 68.1M |
| DFN-MR2 [112] | 3.94 | 19.25 | 1.51 | 14.9M |
| DFN-MR3 | 3.57 | 19.00 | 1.55 | 24.8M |
| IGC-L450M2 [111] | 3.25 | 19.25 | - | 19.3M |
| IGC-L32M26 | 3.31 | 18.75 | 1.56 | 24.1M |
| ResNet pre-activation $L = 65$ $k = 64$ $e = 2$ | 5.26 | 23.24 | - | 1.4M |
| ResNet pre-activation $L = 164$ $k = 64$ $e = 2$ | 4.24 | 19.92 | - | 3.4M |
| ResNet pre-activation $L = 164$ $k = 64$ $e = 4$ | 3.96 | 18.84 | - | 6.8M |
| DenseNet-BC $L = 100$ $k = 12$ $e = 1$ | 4.77 | 22.87 | 1.79 | 0.8M |
| DenseNet-BC $L = 112$ $k = 16$ $e = 1$ | 4.47 | 20.73 | 1.83 | 1.7M |
| DenseNet-BC $L = 130$ $k = 20$ $e = 1$ | 3.86 | 19.62 | 1.84 | 3.4M |
| DenseNet-BC $L = 160$ $k = 24$ $e = 1$ | 3.74 | 18.43 | 1.88 | 6.9M |
| DenseNet-BC $L = 166$ $k = 32$ $e = 1$ | 3.68 | 17.68 | 1.88 | 13.0M |
| DenseNet-BC $L = 190$ $k = 40$ $e = 1$ | 3.75 | 17.22 | 1.79 | 25.8M |
| DenseNet-BC $L = 82$ $k = 8$ $e = 3$ | 4.30 | 21.25 | 1.66 | 0.8M |
| DenseNet-BC $L = 82$ $k = 10$ $e = 4$ | 3.78 | 19.92 | 1.62 | 1.6M |
| DenseNet-BC $L = 88$ $k = 14$ $e = 4$ | 3.57 | 17.68 | 1.55 | 3.5M |
| DenseNet-BC $L = 88$ $k = 20$ $e = 4$ | 3.18 | 16.79 | 1.57 | 7.0M |
| DenseNet-BC $L = 94$ $k = 26$ $e = 4$ | 3.01 | 16.24 | 1.50 | 13.0M |
| DenseNet-BC $L = 118$ $k = 35$ $e = 3$ | 2.99 | 16.18 | 1.50 | 25.7M |
| DenseNet-BC $L = 106$ $k = 33$ $e = 4$ | 2.99 | 15.68 | 1.53 | 25.1M |
| DenseNet-BC $L = 76$ $k = 35$ $e = 6$ | 2.92 | 15.76 | 1.50 | 24.6M |
| DenseNet-BC $L = 64$ $k = 35$ $e = 8$ | 3.13 | 15.95 | 1.50 | 24.9M |

Table 5.4 – Classification error comparison with the state of the art.

more conservative measure of the error rate (on the last iterations) and from statistical differences due to different initialisations and/or due to non-deterministic computations. Still, the coupled ensemble leads to a significantly better performance for all network sizes, even when compared to the reported performance of DenseNet-BC.

Our larger models of coupled DenseNet-BCs (error rates of 2.92% on CIFAR 10, 15.68% on CIFAR 100 and 1.50% on SVHN) perform better than or are on par with all current state of the art implementations that we are aware of at the time of performing these experiments. Only the Shake-Shake S-S-I model [24] performs slightly better on CIFAR 10.

We also compare the performance of coupled ensembles with model architectures that were 'learnt' in a meta learning scenario. The results are presented in § 5.3.9.

### 5.3.7 Ensembles of coupled ensembles

The coupled ensemble approach is limited by the size of the network that can fit into GPU memory and the training time. With the hardware we have access to, it was not possible to go much beyond the 25M parameters. For going further, we resorted to the classical ensembling approach based on independent trainings. An interesting question was whether we could still significantly improve the performance since the classical approach generally plateaus after quite a small number of models and the coupled ensemble approach already includes several branches. For instance, SGDR with snapshots [66] has a significant improvement from 1 to 3 models but not much improvement from 3 to 16 models (see Table 5.4 and 5.5). As doing multiple times the same training is quite costly for large models, we instead ensembled the four large coupled ensemble models, $e = 3, 4, 6, 8$. Results are shown in Table 5.5. We obtained a significant gain by fusing two models and a quite small one from any further fusion of three or four of them. To the best of our knowledge, these ensembles of coupled ensemble networks outperform all state of the art implementations including other ensemble-based ones at the time of these experiments.

### 5.3.8 Parameter usage

Table 5.6, and the lower half of Table 5.4 (rows with $e = 1$ and $e > 1$) highlight the difference in error rates between single and multi branch models. A coupled ensemble model with 13M parameters has an error rate of 16.24% for CIFAR-100. In contrast, none of the single-branch models match this performance even with double the number of parameters. In Table 5.4 we see that coupled ensemble with 13M parameters is better than DenseNet-BC with 25M parameters. In particular

Table 5.5 – Classification error comparison with the state of the art, multiple model trainings.

| Architecture | C10+ | C100+ | SVHN | Params. |
|---|---|---|---|---|
| SGDR WRN-28-10 3 runs × 3 snapshots | 3.25 | 16.64 | - | 329M |
| SGDR WRN-28-10 16 runs × 3 snapshots | 3.14 | 16.21 | - | 1752M |
| DenseNet-BC ensemble of ensembles $e = 6, 4$ | 2.72 | 15.13 | 1.42 | 50M |
| DenseNet-BC ensemble of ensembles $e = 6, 4, 3$ | 2.68 | 15.04 | 1.42 | 75M |
| DenseNet-BC ensemble of ensembles $e = 8, 6, 4, 3$ | 2.73 | 15.05 | 1.41 | 100M |

with 3.5M parameters, coupled ensemble has almost the same performance as the best DenseNet-BC with 25M parameters. This is a parameter reduction of almost 8×. Figure 5.3 compares this with other state of the art models. We see that coupled ensembles are better than most models, with a substantially lesser number of parameters. This means that using this architectural modification we can reduce the model size without compromising on generalisation error. Smaller models are useful for various application such as when deploying models on mobile phone or other embedded systems.

Table 5.6 – Single branch and coupled ensembles CIFAR-100 classification error rate for different number of parameters. The exact architecture details are omitted here. Instead we focus on the error obtained for architectures having the given parameter count.

| Params. | Single-branch | Coupled Ensemble |
|---|---|---|
| 0.8M | 22.87 | 21.25 |
| 1.7M | 20.73 | 19.92 |
| 3.4M | 19.62 | 17.68 |
| 6.9M | 18.43 | 16.79 |
| 13.0M | 17.68 | 16.24 |
| 25.8M | 17.22 | 15.68 |

### 5.3.9 Comparison with *Learnt* architectures

In Table 5.7, we compare the parameter usage and performance of coupled ensembles with model architectures that were recovered using meta learning techniques. We see that our method is competitive with architectures recovered using various learning strategies. The variants which outperform coupled ensembles are using cut-out [15] data augmentation. The results of models with and without cut-out

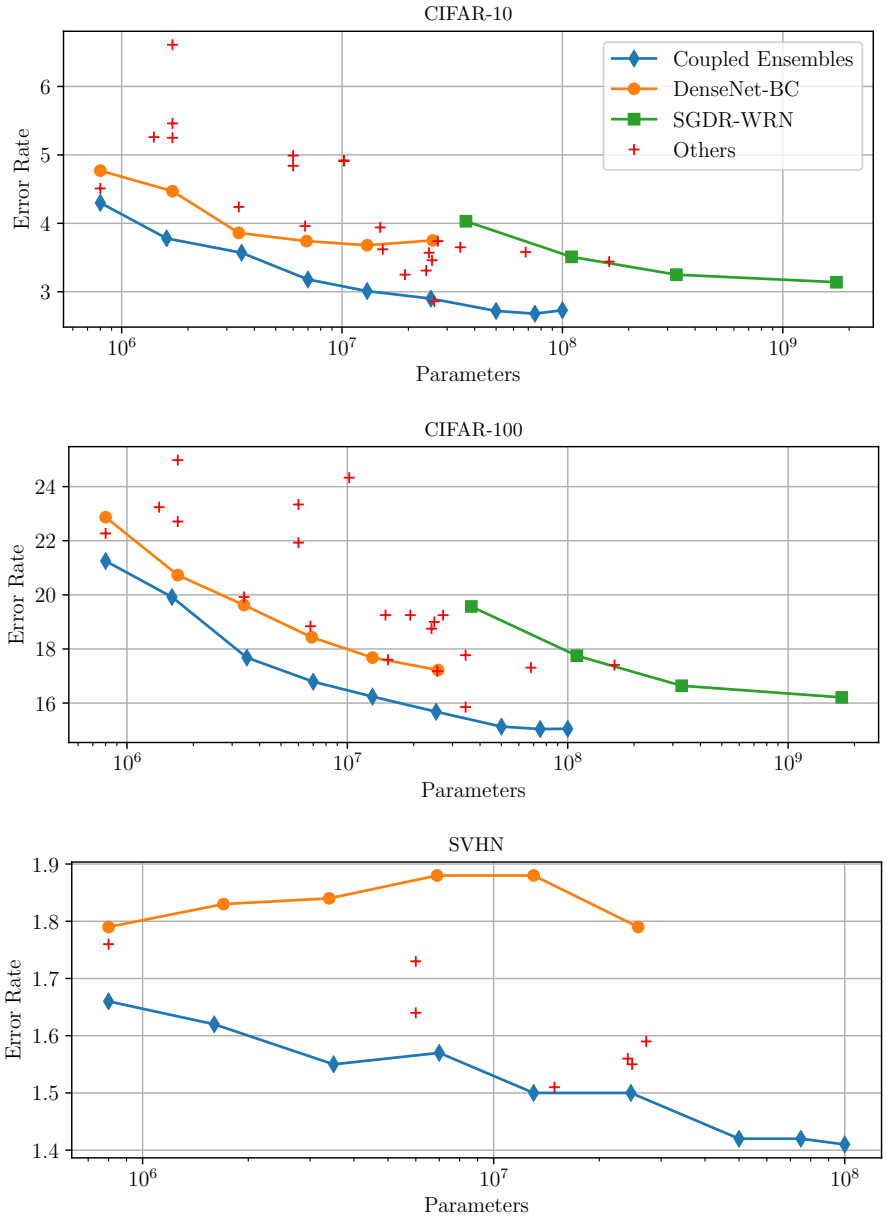



Figure 5.3 – Comparison of parameter usage and error rate among different methods, for CIFAR-10, 100 and SVHN. Coupled ensemble with DenseNet-BC as element block. Single models up to 25M parameters after that points are of ensembles of coupled ensembles; SGDR-WRN: snapshot ensembles up to 110M parameters and ensembles of snapshot ensembles; "Others": all other architectures from Table 5.4, 5.5.

seem to indicate that coupled ensembles can also gain from this data augmentation scheme.

Architecture learning strategies typically define building blocks such as the convolution and maxpool operations, layer connectivity patters, etc. The branching design of coupled ensembles can also be defined as such a building block. Our coupled ensemble idea has been explored in an architecture search setting in [67].

Table 5.7 – Classification error comparison with *learnt* architectures.

| System | C10+ | C100+ | SVHN | #Params |
|---|---|---|---|---|
| Neural Architecture Search v3 [113] | 3.65 | - | - | 37.4M |
| NASNet-A (6 @ 768) [114] | 3.41 | - | - | 3.3M |
| NASNet-A (6 @ 768) + cutout [114] | 2.65 | - | - | 3.3M |
| NASNet-A (7 @ 2304) [114] | 2.97 | - | - | 27.6M |
| NASNet-A (7 @ 2304) + cutout [114] | 2.40 | - | - | 27.6M |
| DARTS (second order) + cutout [64] | 2.83 | - | - | 3.4M |
| DenseNet-BC $L = 82$ $k = 10$ $e = 4$ | 3.78 | 19.92 | 1.62 | 1.6M |
| DenseNet-BC $L = 88$ $k = 14$ $e = 4$ | 3.57 | 17.68 | 1.55 | 3.5M |
| DenseNet-BC $L = 76$ $k = 35$ $e = 6$ | 2.92 | 15.76 | 1.50 | 24.6M |

### 5.3.10 Preliminary experiments on ImageNet

We conducted preliminary experiments on ILSVRC2012 [87] to compare single branch models and multi branch coupled ensembles.

For a baseline single branch model, we use DenseNet-169-k32-e1 and DenseNet-201-k32-e1, and we compare this with coupled ensemble DenseNet-121-k30-e2. We realise this is not a state-of-the-art baseline but due to the constraints we had this was the strongest possible. The current results are in Table 5.8. It is hard to draw conclusions from these results and there is a need to try models of different sizes to have a proper conclusion. Since ILSVRC2012 is a very large scale data set it is in general quite hard to find differences among methods. One interesting extension would be to use the full ImageNet data set instead of of the ILSVRC subset. The full data set is quite unbalanced among categories and is more challenging for all methods.

## 5.4 Training efficiency

Coupled ensembles offer a large reduction in the number of parameters needed to obtain a desired error rate, as discussed in § 5.3.8. In this section we investigate

Table 5.8 – Preliminary results on ImageNet.

| L | k | e | Params. | Epochs | Top-1 error |
|---|---|---|---|---|---|
| 169 | 32 | 1 | 14.2M | 90 | 23.90 |
| 201 | 32 | 1 | 20.0M | 90 | 23.29 |
| 201 | 22 | 2 | 19.7M | 90 | 23.79 |
| 167 | 27 | 2 | 20.9M | 90 | 23.74 |

if this reduction in parameters also translates to faster training, and reduction in overfitting. We compare the training times for different parameter budgets and explore overfitting when the number of labelled samples is low.

### 5.4.1 Training time budget

Our discussion and comparison between coupled ensembles and single branch models has been in the context of a single fixed parameter budget between the two types of models. In this context we have shown that coupled ensembles provide a superior performance as compared to a single branch model with the same number of parameters. However, the training time of coupled ensembles is longer and increases with the number of branches. Though this might be improved by better parallelisation, we investigate here whether the multi branch coupled ensembles can still improve over a single branch model with a fixed training time budget, with the current implementation.

There are several approaches to reducing the training time: (i) reduce the number of training epochs; (ii) reduce the parameter count, smaller models will have shorter forward and backward pass times; (iii) increase the width while reducing the depth to keep the parameters constant. Wider models, upto a limit, are more efficiently computed by current hardware.

Table 5.9 shows the results obtained for these three options after training on CIFAR-10 and CIFAR-100. The baseline is a single branch DenseNet-BC $L = 190$, $k = 40$, $e = 1$, which is trained for 300 epochs and needs 80 hours of wall clock time. The corresponding multi branch baseline with the same parameter budget is DenseNet-BC $L = 106$, $k = 33$, $e = 4$, whose training wall clock time is 127 hours for the same 300 epochs, which is about $1.6\times$ that of the baseline.

We can limit the training wall clock time to 80 hours, in which we can train the coupled ensemble model for 188 epochs. We see in Table 5.9 that this obtains an error rate that is better than the single branch model. We can also try different architectural configurations and train for the full 300 epochs in 80 hours. We see from the table that these options achieve better error rates (row 4 and 5). These options correspond to reducing the number of parameters and having a

wider network respectively. In one case we reduce the depth, and in the other we increase the width of the network while decreasing the width to keep the number of parameters constant. We go further and reduce the number of parameters further and we see that these architectures also achieve better performance than an single branch model.

We can conclude that while strictly equivalent coupled ensembles might have a higher training time, it is possible to have configurations that match the performance of single branch models. Moreover these configurations might actually be faster to train without sacrificing any performance. This reinforces the efficacy of coupled ensembles by showing that we can have smaller models that are fast to train and which perform at par with target baselines. As an example, we have DenseNet-BC $L = 88$, $k = 20$, $e = 4$, which has an error rate better than the single branch baseline with a parameter count reduced by 3.7 and a training time reduced by 1.5.

Table 5.9 – Error rate versus training time and parameters. All times are measured on one GTX 1080Ti.

| Runs | L | k | e | Params. | Epochs | Train time (h) | C10+ | C100+ |
|---|---|---|---|---|---|---|---|---|
| 10 | 190 | 40 | 1 | 25.8M | 300 | 79.3 | 3.75±0.17 | 17.31±0.16 |
| 10 | 106 | 33 | 4 | 25.1M | 300 | 126.6 | 2.99±0.05 | 15.80±0.16 |
| 10 | 106 | 33 | 4 | 25.1M | 188 | 79.3 | 3.15±0.06 | 16.22±0.08 |
| 5 | 88 | 33 | 4 | 18.6M | 300 | 79.9 | 3.05±0.06 | 16.09±0.15 |
| 5 | 76 | 44 | 4 | 25.8M | 300 | 79.9 | 3.05±0.09 | 16.06±0.17 |
| 5 | 94 | 26 | 4 | 13.0M | 300 | 67.1 | 3.13±0.09 | 16.27±0.16 |
| 5 | 88 | 20 | 4 | 7.0M | 300 | 52.3 | 3.32±0.04 | 16.87±0.15 |

### 5.4.2 Training data budget

In this section we compare the performance of single branch models and coupled ensembles in the context of a low labelled training data scenario. We train a single branch model and multi branch coupled ensemble, both having the same number of parameters, on two datasets: STL-10 [10] and a 10K balanced random subset of the 50K CIFAR-100 training set. STL-10 comprises images of size 96x96, grouped into 10 classes. Each class has 500 training images. The test set has 8000 images. Since STL has less training examples as compared to the test set, it is critical for the model to not overfit to the training set. Results are shown in Table 5.10. We see that for a fixed parameter budget, coupled ensembles significantly outperform the single branch model.

Table 5.10 – Results with low training data and limited labels.

| Dataset | Single-Branch | Coupled Ensemble | Params. |
|---|---|---|---|
| Low data | | | |
| STL-10 | 41.73 | 32.01 | 0.8M |
| CIFAR-100 10K subset | 41.84 | 36.30 | 3.2M |
| Limited labels | | | |
| CIFAR-100, 5K labels | 60.63 | 60.46 | 0.8M |
| CIFAR-100, 10K labels | 46.81 | 41.58 | 0.8M |
| CIFAR-100, 20K labels | 38.93 | 36.12 | 0.8M |
| CIFAR-100, 40K labels | 30.91 | 27.36 | 0.8M |
| CIFAR-10, 5K labels | 17.20 | 15.01 | 0.8M |
| CIFAR-10, 10K labels | 12.89 | 11.58 | 0.8M |
| CIFAR-10, 20K labels | 8.14 | 7.75 | 0.8M |
| CIFAR-10, 40K labels | 5.49 | 4.94 | 0.8M |

## 5.5 Implementation details

Figure 5.1 shows the common structure of the test (top) and train (bottom) versions of networks used as element blocks. In Figure 5.2b shows the possible positions for the fuse layer, after the `fc` layer or after the `sm` layer. Figure 5.2a shows the fuse layer positions during training of coupled ensembles.

We reuse existing convolutional network architectures as "element blocks" in their original form as much as possible both for efficiency and for ensuring more meaningful comparisons.

Each of the $e$ branches is defined by a parameter vector $W_e$ containing the same parameters as the original implementation. The global network is defined by a parameter vector $W$ which is a concatenation of all the $W_e$ parameter vectors. When training is done in the coupled mode and the prediction is done in a separate mode or vice-versa, a dedicated script is used for splitting the $W$ vector into the $W_e$ ones or vice-versa. In all coupled ensemble models, for all train versions and for all test versions, the same global parameter vector $W$ is used with the same split and defining the same element block functions. This is how we can combine in any way all of the four possible training conditions with all the three possible prediction conditions, even though not all of them are consistent or equally efficient.

The overall network architecture is determined by:

- A global hyper-parameter specifying the train versus test mode;

- The global hyper-parameter $e$ specifying the number of branches;
- The global hyper-parameter specifying after which layer the fuse layer should be placed (`FC`, `LSM`, `LL`);
- either one element block to be replicated $e$ times with its own hyper-parameters or a list of $e$ element blocks, each with its own hyper-parameters. In our experiments, we have used the same hyper parameters for all the element blocks.

### 5.5.1 Test time equivalence between FC average and LogSoftMax average

Given branches $E = \{E_1, E_2, ..E_e\}$, each $E_i$ produces a score vector of dimension $C$, where $C$ is the number of categories. An element of $E_i$ is referenced as $E_i^c$, where $c \in [1, C]$. FC_Average denotes averaging the raw activations from each branch. LSM_Average denotes averaging across branches, after a `log_softmax` operation in applied on each branch activation vector, separately.

Case 1: FC_average: $Scores_{FC}^c = \sum_{i=1}^{e} E_i^c$

Case 2:

$$\begin{aligned} LogSoftMax(E_n^c) &= \log \frac{\exp(E_e^c)}{\sum_c \exp(E_e^c)} \\ &= \log \exp(E_e^c) - \log \sum_c \exp(E_e^c) \\ &= E_e^c - Z_e \end{aligned} \tag{5.1}$$

LSM_average: $Scores_{LSM}^c = \sum_{i=1}^{e} E_i^c - \sum_{i=1}^{e} Z_i$, where $Z_e = \log \sum_C \exp(E_e^c)$. We see that the LSM_average score vector is a translated version of the FC_average score vector. Also, doing an arithmetic average of `log_softmax` values is equivalent to doing a geometric average of `softmax` values. This holds during inference where we are interested only in the maximum value.

### 5.5.2 Micro-batch versus mini-batch

For some of the larger models, it was not possible to train them with a (mini-)batch size of 64. In this case, we split data batches into $m$ "micro-batches" with $b/m$ samples each, $b$ being equal to batch size. We accumulate the gradient over these micro batches and take the average over the $m$ micro-batches to get an almost equivalent gradient, as we would have got if we processed data directly as a single batch.

The gradient would have been exactly equivalent but for the `batchnorm` layer. This is because BatchNorm uses the batch mean and standard deviation to normalise the activations during the forward pass. This means that the micro-batch statistics are used whereas, ideally to get an exact equivalent, we would need the whole batch statistics. However, in practice this does not make a significant difference. Hence, to have the same settings for comparison among different models, we perform parameter updates using gradient for a batch, while performing forward passes with micro-batches (to have an optimal throughput).

In the single-branch case for a given parameter budget, the need for memory depends mostly on the network depth and on the mini-batch (or micro-batch) size. We use the micro-batch "trick" for adjusting the memory need to what is available while keeping the mini-batch size equal to the default value (64 for CIFAR and SVHN). Though this is not strictly equivalent, this does not hurt performance and this would even yield a slight increase.

The multi-branch version does not require more memory if the branches' width is kept constant. More memory is needed only when the branches' width is reduced. Indeed, if we use branches with a constant parameter budget, we have to reduce either the width or the depth or both. We did hyper-parameter search experiments by cross-validation and these indicated that the best option was a reduction of both while the exact trade-off was not very critical. In practice, for our "full-size" experiments ( 25M parameters) we did the training within the 11GB memory of a GTX 1080 Ti card, using micro-batch sizes of 16 for the single-branch versions and of 8 for multi-branch ones. Splitting the network over two GPU cards allows for doubling the micro-batch sizes. However, this usually does not significantly increases the speed and this does not lead to a performance improvement.

## 5.6 Discussion on related work

**Multi-column architectures** The network architecture that we propose has similarities with Neural Networks Committees [9] and Multi Column Deep Neural Network (MCDNN) [8], which are a type of ensemble of networks where the "committee members" are the "DNN columns". These correspond to our element blocks (or branches). However, our coupled ensemble networks differ in the following aspects: (i) we train a *single* model which is *composed* of branches, while they train each member or column *separately*. (ii) we have a fixed parameter budget for the *entire* model for improving the performance. This is contrary to improving it by utilising multiple models of fixed size and therefore increasing the overall size (though both are not exclusive); (iii) we combine the activations of the branches by combining their log-probabilities over the target categories, and (iv) we used the same input for all branches while they considered different pre-processing (data

augmentation) blocks for different members or different subsets of columns.

**Multi-branch architectures** Multi-branch architectures such as Inception [100] and ResNet [35] have been very successful in several different tasks. Recently, modifications have been proposed [107, 7] for these architectures using the concept of "grouped convolutions", in order to factorise spatial and depth wise feature extraction. These modifications additionally advocate the use of *template* building blocks stacked together to form the complete model. This modification is at the level of the *building blocks* of their corresponding *base* architectures: ResNet and Inception respectively. In contrast we propose a generic modification of the structure of convolutional networks at the global architecture level. This includes a template in which the specific architecture of an "element block" is specified, and then this "element block" is replicated as parallel branches to form the final composite model, which are then trained *jointly*.

To further improve the performance of such architectures, Shake-Shake regularisation [24] proposes a stochastic mixture of each of the branches and has achieved good results on the CIFAR data sets. However, the number of epochs required for convergence is much higher compared to the base model. Additionally, the technique seems to depend on the batch size. In contrast, we apply our method using the exact *same* hyper-parameters as used in the underlying CNN.

[112] investigates the usage of parallel paths in a ResNet, connecting layers across paths to allow information exchange between them. However this requires modification at a local level of each of the residual blocks. In contrast, our method is a generic re-arrangement of a given architecture's parameters, which does not introduce additional choices. Additionally, we empirically confirm that our proposed configuration leads to an efficient usage of parameters.

**Neural network ensembles** Ensembling is a reliable technique to increase the performance of models for a task. Due to the presence of several local minima [50], multiple trainings of the exact same neural network architecture can reach a different distribution of errors on a per-class basis. Hence, combining their outputs lead to improved performance on the overall task. This was observed very early [34] and is now commonly used for obtaining top results in classification challenges, despite the increase in training and prediction cost. Our proposed model architecture is not an ensemble of independent networks given that we have a single model made up of parallel branches that is trained jointly. This is similar in spirit to the `residual block` in ResNet and ResNeXt, and to the `inception` module in Inception *but it is done at the global network level*. We would like to emphasise here that "arranging" a given budget of parameters into parallel branches leads to an increase in performance (see Table 5.1, 5.2, 5.3, 5.4). Additionally, the classical

ensembling approach can still be applied for the fusion of independently trained coupled ensemble models, where it leads to a significant performance improvement (see Table 5.5).

Training multiple models for ensembling can be both time and resource heavy. To alleviate this, recent work has proposed reusing model weights obtained during the training of a single model. Snapshot ensembles [46] and training with restarts [66] used the ensembling approach on checkpoints during the training process instead of using fully converged models. This approach is quite efficient since the obtained performance is higher for a fixed training time budget. However, both the overall model size and prediction time are significantly increased. Given a model size and performance measure, our approach aims to either keep the model size constant and improve the performance, or to obtain the same performance with a smaller model size.

**Model pruning** Model pruning techniques aim to reduce the size of trained models. Coupled ensembles provide a reduction in the number of parameters for a target test error rate, which leads to smaller model size. After training has completed, various pruning techniques can be used for further reduction in the model size. These techniques are based on weight magnitude [57, 33] and on activation values [60]. In these cases pruning is done *a posteriori*, whereas in the case of coupled ensembles it is an *a priori* step. This leads to coupled ensembles having a smaller architecture during training which reduces the computation requirements for training as well. Pruning techniques may be applicable to coupled ensembles after they finish training but we have not investigated this. For completeness, we should point out that there exists other architectures which are aimed towards smaller model sizes such as MobileNet [44], and recently architectures optimised for having a small model size are also being *searched* for [101].

## 5.7 Conclusion

In this chapter we introduced *coupled ensembles*, a new convolutional network architecture and training framework. We evaluated the model on image classification tasks and compared it with current state of the art architectures. Our experiments showed that the parallel branches of coupled ensembles give better performance as compared to single branch models. Given a parameter budget, it is more efficient to have the parameters distributed over parallel branches rather than having a single branch. For a certain target performance, the coupled ensemble approach is able to achieve it with much fewer parameters, leading to smaller and more efficient models. CNNs are an integral part of several machine learning systems and improvements in their design will have a positive impact on all these systems.

# Chapter 6

# Conclusion and Perspectives

In this chapter, we summarise the contributions for this thesis. For each contribution we provide some extensions and future directions of work.

## 6.1 Hierarchical classification

In Chapter 3, we explored the adaptation of classifiers with respect to change in the label distribution of a data set. We saw that it is possible to use a semantic hierarchy of labels to improve the classification performance. This was achieved by making the predicted probabilities consistent for each parent and child nodes in the label tree.

At present, we have considered only one level of semantic relationships: a coarse category, which is a parent node, and the corresponding fine categories, which form the child nodes. It is possible to extend this procedure to a semantic hierarchy tree where we have more than two levels. In that case we will need to ensure that the prediction probabilities are consistent throughout the path from a parent node to the child nodes. Having such a tree, and classifiers corresponding to each semantic level offers the possibility to dynamically choose the granularity of classification. For example, we can have relatively weak classifiers trained at coarse levels and progressively more powerful classifiers at the finer levels. This will help to make a trade-off between computation cost and prediction accuracy.

Currently, our prediction probability relies on the fact that the semantic hierarchy is provided to us but all data sets do have this meta-information. However it maybe possible to *learn* a hierarchy from a given set of classes. One way to jointly learn a hierarchy along with the classifier is to group together categories that have similar responses. The response can be computed at the last convolution layer before the final classifier output. Another way would be to construct a confusion matrix of the categories and group together categories that are are predicted to be

the same. If we have a confusion matrix with rows as fine categories, and columns as coarse categories, we assign to each fine category the coarse category which they are most predicted to belong to. These steps can be repeated iteratively until the assignments become stable. It will be interesting to see the nature of the *learnt* hierarchy. We will have to compare this to the original hierarchies available and analyse the effect it has on the prediction probability adjustment procedure.

## 6.2 Continual learning with replay buffers

We investigated the use of replay buffers in continual learning in Chapter 4. First we evaluated the state of current GAN models for usage as a replay buffer. Our experiments indicate that there is a substantial gap in the quality and diversity of samples from a GAN model as compared to real data samples. Currently, we have proposed post-processing steps to select good samples out of all samples from a GAN. The next step will be to consider what architectural and training modification can be done to enable better GAN models. We saw the first step through our usage of multiple GANs to cover an increased portion of the data distribution.

The next step will be to train different GAN models on different data samples. This will be an approach similar to boosting [23]. We will extract features from a pre-trained classifier, and compute distance between the features of real samples and generated samples. We hypothesise that larger distances in feature space implies that the GAN model has difficulties in modelling the corresponding categories. Similar to boosting where harder samples are selected for further training, we can train specific GANs on such 'hard' categories.

We further analysed training a neural network on several tasks sequentially, in a continual learning setting. We analysed the learnt representation and saw that this manner of training led to worse performance as compared to training on all tasks at once. Training on all tasks has benefits of multi-task learning [6] which might lead to better performance. However we also saw that there is a high loss barrier in the weight space defined by the two learnt representations. This indicates the need for alternate training procedures, possibly with additional loss functions or modification to backpropagation when computing the gradients.

## 6.3 Coupled ensembles

Coupled ensembles was introduced as a new neural network architecture and was presented in Chapter 5. Coupled ensembles enables training of classifiers with a substantially reduced parameter count. This makes the trained models smaller,

which helps in training and during deployment, in resource constrained environments. Currently we have evaluated this architecture for image classification tasks. Since CNNs are used for learning representations in other tasks such as objection detection and semantic segmentation, future work needs to evaluate the effectiveness of coupled ensembles in these tasks as well. In this case the implementation is straightforward since the features computed by coupled ensembles can be directly passed to the specific modules used for these tasks. It is not trivial to apply the coupled ensemble framework to sequence based tasks, such as language modelling and translation. We need to decide how each branch output needs to be combined so as to be consistent with the sequential aspect of the problem. However we can more easily see the effect in case of convolutional `seq2seq` models [25].

An important area of research is in predictive uncertainty of neural network outputs. This involves calibration of the probability output which means that the magnitude should correspond to it being correct or incorrect. Normally if an incorrect prediction is made with high probability, it is an indication of poor calibration. Current neural network models have been shown to suffer from this issue [31]. Lakshminarayanan, Pritzel, and Blundell [56] have shown that separately trained ensembles of neural networks can be used to improve the predictive uncertainty of neural networks. Our design is readily able to provide this benefit since we can extract the predictions from each of our branches individually. This makes coupled ensembles usable for estimating the prediction uncertainty, in addition to being good classifiers. However, we need to do further detailed studies to evaluate this.

The coupled ensemble model also shares resemblance to models proposed for adversarial robustness [75] and privacy preserving training [76]. These issues are relevant and important today because of the wide scale deployment of machine learning systems across all sectors of society. The design of coupled ensembles provides opportunities for exploring the relevance and effectiveness in these aspects. This is a future line of research where coupled ensembles can be extended and evaluated for these aspects.

## 6.4 General perspectives

Deep learning models have achieved success in a variety of fields but there still remains a lot of room for improvement. Continual learning algorithms are trying to make models more general and universal with respect to the number of tasks that can be trained on a single model. However, during inference mode, most proposed methods need to be informed about the task which the test sample belongs to. Future research should focus on methods which can accommodate several tasks and be able to infer the task of a test sample. In order to promote such

techniques, the community needs to develop benchmarks and evaluation metrics specifically focused at this. Currently continual learning does not yet have a standard evaluation framework. The beginnings of a such a framework are starting to be developed. New evaluation frameworks should explicitly pay attention to the size of the composite model, the task inference mechanism, and training time needed for adding each additional task to a network. There is a need to go beyond the average accuracy over tasks that is the current pre-dominant performance metric.

The *lottery ticket hypothesis* [22] has shown that for various networks with a large number of parameters, there exists substantially smaller sub-networks for which the performance is the same as the large network. One approach towards continual learning is through learning universal representations, as discussed in Chapter 4. Using the lottery ticket hypothesis is one viable way to find task specific sub-networks inside a large network, in a principled manner. Starting from a large network, we can identify a small sub-network for each task consecutively. We will have a limit on the number of tasks that can be accommodated inside one network, depending on the starting size of the network and the nature of the tasks.

# List of Publications


1. Anuvabh Dutt. "Towards Incremental Learning with Deep Convolutional Networks". In: *Conférence en Recherche d'Informations et Applications - CORIA 2017, 14th French Information Retrieval Conference, Marseille, France, March 29-31, 2017. Proceedings.* 2017, pp. 385–394. URL: https://doi.org/10.24348/coria.2017.RJCRI%5C_4

2. Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Improving image classification using coarse and fine labels". In: *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval.* ACM. 2017, pp. 438–442

3. Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Improving Hierarchical Image Classification with Merged CNN Architectures". In: *Proceedings of the 15th International Workshop on Content-Based Multimedia Indexing, CBMI 2017, Florence, Italy, June 19-21, 2017.* 2017, 31:1–31:7

4. Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled ensembles of neural networks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings.* OpenReview.net, 2018. URL: https://openreview.net/forum?id=rylRCUJDG

5. Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled Ensembles of Neural Networks". In: *2018 International Conference on Content-Based Multimedia Indexing (CBMI).* IEEE. 2018, pp. 1–6

6. Stefen Chan Wai Tim, Michele Rombaut, Denis Pellerin, and Anuvabh Dutt. "Descriptor extraction based on a multilayer dictionary architecture for classification of natural images". In: *Computer Vision and Image Understanding* (2018). URL: https://doi.org/10.1016/j.cviu.2018.08.002

7. Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled ensembles of neural networks". In: *Neurocomputing* (2019). URL: https://doi.org/10.1016/j.neucom.2018.10.092

8. Pham Tanh Dat, Anuvabh Dutt, Denis Pellerin, and Georges Quénot. “Classifier training from a generative model”. In: *2019 International Conference on Content-Based Multimedia Indexing (CBMI)*. IEEE. 2019

# Appendix A

# Reproduciblity Issue

Our experiments in Chapter 5 mentioned that we were not able to reproduce the results of a paper exactly, even when using the hyper-parameter values mentioned in the original papers. Investigating this issue raised several questions about the source of randomness in results. We compared the results of training several neural network models if different frameworks with various setting for random seed and other computational settings.

## A.1 Performance measurement and reproducibility issues

When attempting to compare the relative performance of different methods, we face the issue of the reproducibility of the experiments and of the statistical significance of the observed difference between performance measures. Even for the same experiment, we identified the five following sources of variation in the performance measure:

- Underlying framework for the implementation: we made experiments with Torch7 (lua) and with PyTorch.
- Random seed for the network initialisation.
- CuDNN non-determinism during training: GPU associative operations are by default fast but non-deterministic. We observed that the results varies even for a same tool and the same seed. In practice, the observed variation is as important as when changing the seed.
- Fluctuations associated to the computed moving average and standard deviation in batch normalisation: these fluctuations can be observed even when

training with the learning rate, the SGD momentum and the weight decay all set to 0. During the last few epochs of training, their level of influence is the same as with the default value of these hyper-parameters.

- Choice of the model instance chosen from training epochs: the model obtained after the last epoch, or the best performing model. Note that choosing the best performing model involves looking at test data.

Regardless of the implementation, the numerical determinism, the Batch Norm moving average, and the epoch sampling questions, we should still expect a dispersion of the evaluation measure according to the choice of the random initialisation since different random seeds will likely lead to different local minima. It is generally considered that the local minima obtained with "properly designed and trained" neural networks should all have similar performance [50]. We do observe a relatively small dispersion (quantified by the standard deviation below) confirming this hypothesis. This dispersion may be small but it is not negligible and it complicates the comparisons between methods since differences in measures lower than their dispersions is likely to be non-significant. Classical statistical significance tests do not help much here since differences that are statistically significant in this sense can be observed between models obtained just with different seeds (and even with the same seed), everything else being kept equal.

Experiments reported in this section gives an estimation of the dispersion in the particular case of a moderate scale model. We generally cannot afford doing a large number of trials for larger models.

We tried to quantify the relative importance of the different effects in the particular case of DenseNet-BC with $L = 100, k = 12$ on CIFAR 100. The upper part of Table A.1 shows the results obtained for the same experiment in the four groups of three rows. We tried four combinations corresponding to the use of Torch7 versus PyTorch and to the use of the same seed versus the use of different seeds. For each of these configuration, we used as the performance measure: (i) the error rate of the model computed at the last epoch or (ii) the average of the error rate of the models computed at the last 10 epochs, (iii) the error rate of the model having the lowest error rate over all epochs. For these $2\times2\times3$ cases, we present the minimum, the median, the maximum and the mean$\pm$standard deviation over 10 measures corresponding to 10 identical runs (except for the seed when indicated). Additionally, in the case of the average of the error rate of the models computed at the 10 last epochs, we present the root mean square of the standard deviation of the fluctuations on the last 10 epochs (which is the same as the square root of the mean of their variance). We make the following observations:

- There does not seem to be a significant difference between Torch7 and PyTorch implementations;

- There does not seem to be a significant difference between using a same seed and using different seeds; the dispersion observed using the same seed (with everything else being equal) implies that there is no way to exactly reproduce results;

- There does not seem to be a significant difference between the means over the 10 measures computed on the single last epoch and the means over the 10 measures computed on the last 10 epochs;

- The standard deviation of the measures computed on the 10 runs is slightly but consistently smaller when the measures are computed on the last 10 epochs than when they are computed on the single last epoch; this is the same for the difference between the best and the worst measures; this was expected since averaging the measure on the last 10 epochs reduces the fluctuations due to the moving average and standard deviation computed in batch normalization and possibly too the the random fluctuations due to the final learning steps;

- The mean of the measures computed on the 10 runs is significantly lower when the measure is taken at the best epoch than when they are computed either on the single last epoch or on the last 10 epochs. This is expected since the minimum is always below the average. However, presenting this measure involves using the test data for selecting the best model.

Following these observations, we propose a method for ensuring the best reproducibility and the fairest comparisons. Choosing the measure as the minimum of the error rate for all models computed during the training seems neither realistic nor a good practice since we have no way to know which model will be the best one without looking at the results (cross-validation cannot be used for that) and this is like tuning on the test set. Even though this is not necessarily unfair for system comparison if the measures are done in this condition for all systems, this does introduce a bias for the absolute performance estimation. Using the error rate at the last iteration or at the 10 last iteration does not seem to make a difference in the mean but the standard deviation is smaller for the latter, therefore this one should be preferred when a single experiment is conducted. We also checked that using the 10 or the 25 last epochs does not make much difference (learning at this point does not seem to lead to further improvement). A value different from 10 can be used and this is not critical. In all the CIFAR experiments reported in this paper, we used the average of the error rate for the models obtained at the last 10 epochs as this should be (slightly) more robust and more conservative. The case for SVHN experiments is slightly different since there is a much smaller number of much bigger epochs; we used the last 4 iterations in this case.

Table A.1 – Performance measurement and reproducibility issues. Statistics on 10 runs.

| Seeds | Impl. | Last | $L$ | $k$ | $e$ | Min. | Med. | Max. | Mean±SD | RMS(SD) |
|---|---|---|---|---|---|---|---|---|---|---|
| diff. | PyT. | 1 | 100 | 12 | 1 | 22.64 | 22.80 | 23.22 | 22.89±0.21 | n/a |
| diff. | PyT. | 10 | 100 | 12 | 1 | 22.67 | 22.83 | 23.14 | 22.87±0.17 | 0.13 |
| diff. | PyT. | best | 100 | 12 | 1 | 22.13 | 22.56 | 22.91 | 22.54±0.24 | n/a |
| same | PyT. | 1 | 100 | 12 | 1 | 22.77 | 23.05 | 23.55 | 23.06±0.23 | n/a |
| same | PyT. | 10 | 100 | 12 | 1 | 22.81 | 22.98 | 23.49 | 23.04±0.22 | 0.11 |
| same | PyT. | best | 100 | 12 | 1 | 22.44 | 22.67 | 23.02 | 22.71±0.18 | n/a |
| diff. | LuaT. | 1 | 100 | 12 | 1 | 22.55 | 22.94 | 23.11 | 22.90±0.20 | n/a |
| diff. | LuaT. | 10 | 100 | 12 | 1 | 22.55 | 22.89 | 23.08 | 22.86±0.20 | 0.12 |
| diff. | LuaT. | best | 100 | 12 | 1 | 22.17 | 22.52 | 22.75 | 22.49±0.18 | n/a |
| same | LuaT. | 1 | 100 | 12 | 1 | 22.33 | 22.82 | 23.58 | 22.82±0.34 | n/a |
| same | LuaT. | 10 | 100 | 12 | 1 | 22.47 | 22.92 | 23.51 | 22.87±0.30 | 0.12 |
| same | LuaT. | best | 100 | 12 | 1 | 22.24 | 22.51 | 23.24 | 22.54±0.29 | n/a |
| diff. | PyT. | 1 | 82 | 8 | 3 | 21.27 | 21.44 | 21.70 | 21.49±0.15 | n/a |
| diff. | PyT. | 10 | 82 | 8 | 3 | 21.24 | 21.46 | 21.63 | 21.45±0.11 | 0.12 |
| diff. | PyT. | best | 82 | 8 | 3 | 20.84 | 21.18 | 21.30 | 21.14±0.14 | n/a |
| diff. | PyT. | 1 | 100 | 12 | 4 | 17.24 | 17.71 | 17.86 | 17.65±0.18 | n/a |
| diff. | PyT. | 10 | 100 | 12 | 4 | 17.37 | 17.67 | 17.81 | 17.66±0.14 | 0.11 |
| diff. | PyT. | best | 100 | 12 | 4 | 17.11 | 17.46 | 17.66 | 17.45±0.16 | n/a |
| diff. | PyT. | 1 | 106 | 33 | 4 | 15.59 | 15.69 | 16.02 | 15.75±0.15 | n/a |
| diff. | PyT. | 10 | 106 | 33 | 4 | 15.64 | 15.78 | 16.18 | 15.80±0.16 | 0.10 |
| diff. | PyT. | best | 106 | 33 | 4 | 15.35 | 15.59 | 15.94 | 15.57±0.17 | n/a |

These observations have been made in a quite specific case but the principle and the conclusions (use of the average of the error rate from the last epochs should lead to more robust and conservative results) are likely to be general. Table A.1 also shows the results for a coupled ensemble network of comparable size, for a coupled ensemble network four times bigger, and for a coupled ensemble network approximately 32 times bigger. Similar observations can be made and, additionally, we can observe that both the range and the standard deviations are smaller for networks of comparable sizes. This might be because an averaging is already made between the branches leading to a reduction of the variance.

# Appendix B

# Multiple data set training

In Chapter 4 we performed an analysis on the various components of training in a continual learning scenario. The tasks were created by splitting categories of the CIFAR data sets. A more drastic change in tasks can occur when each of the tasks have different distributions of the input values, such as when one task is the gray scale MNIST and another task is the colour CIFAR. We perform some experiments in this section to highlight this effect and show that there is a need for specialised techniques to ensure that the input data distribution is similar.

## B.1 Training on multiple datasets

We train a ResNet-20 architecture to jointly classify CIFAR-10 and CIFAR-100. Following the previous methodology, we have variants for training: Train jointly on both data sets, and train incrementally one after the other. Joint training gives and error of 23.82% and training incrementally (first CIFAR-10 and then CIFAR-100) gives error of 24.02%. Weight space analysis shows that again there is a high barrier between the two solutions.

**Training on datasets with same semantics, different data distribution** We consider two digit datasets, MNIST and SVHN. MNIST is a grayscale dataset whereas SVHN consists of colour images. Both these data sets have the same categories, the digits 0 to 9.

To train a model on both these data sets, we need to unify the data format. We replicate the MNIST data along the channel dimension. This gives MNIST digits with 3 channels. Currently pad the images to 32x32 for compatibility with CIFAR experiments. For SVHN we are only using the 'train' split and not using the 'extra' split.

We first train a network on SVHN. This model achieves 4.11% error on the

SVHN test set. Using this model on the MNIST test set, we get an error of 25.23%. While this is not a great error rate for MNIST, it is much better than random predictions.

We also train the same model on MNIST, and it obtain an error rate of 0.31%. Using this model on the SVHN test set, we get an error of 89.97%, which is similar to random predictions. The possible reason for this training on MNIST alone is not enough to capture the semantics of SVHN because SVHN images are colour images and have greater variety in their spatial orientation.

We next incorporate each data set into a model trained on the other. Adding MNIST to a model trained on SVHN gives error rates 4.18% and 0.80% for SVHN and MNIST respectively. Adding SVHN to a model trained on MNIST gives error rates of 3.74% and 0.87% on SVHN and MNIST respectively. Individual double training gives error rate of 3.62% on SVHN and 0.66% on MNIST. Interesting: the result is better or comparable on both data sets after inclusion of the other one.

**Colour MNIST** SVHN and MNIST differ in the fact that one has colour images while the other has greyscale images. We make a variant of MNIST which is colour and call it Colour MNIST. First we create a mask for the background and a mask for the image pixels. For each of the 3 channels in each mask we pick a random value between 0 and 1. This creates a randomised colour image of the digits (Figure B.1 shows some samples). We first train a model on this colour MNIST data set, where it achieves a test error of 0.41%. Using this model on the SVHN test set, we get an error of 42.16% which is a significant improvement over 89.97% which was obtained from the model trained on greyscale MNIST.

Figure B.1 – Sample images from Colour MNIST

# Bibliography


[1] Rahaf Aljundi, Punarjay Chakravarty, and Tinne Tuytelaars. "Expert gate: Lifelong learning with a network of experts". In: *CVPR* (2017).

[2] Bernard Ans and Stéphane Rousset. "Avoiding catastrophic forgetting by coupling two reverberating neural networks". In: *Comptes Rendus de l'Académie des Sciences-Series III-Sciences de la Vie* 320.12 (1997), pp. 989–997.

[3] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016). URL: http://arxiv.org/abs/1607.06450.

[4] Andrew Brock, Jeff Donahue, and Karen Simonyan. "Large Scale GAN Training for High Fidelity Natural Image Synthesis". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=B1xsqj09Fm.

[5] Noam Brown and Tuomas Sandholm. "Superhuman AI for heads-up no-limit poker: Libratus beats top professionals". In: *Science* 359.6374 (2018), pp. 418–424.

[6] Rich Caruana. "Multitask learning". In: *Machine learning* 28.1 (1997), pp. 41–75.

[7] François Chollet. "Xception: Deep learning with depthwise separable convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1251–1258.

[8] Dan Ciresan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 2012, pp. 3642–3649.

[9] Dan Ciresan et al. "Convolutional neural network committees for handwritten character classification". In: *2011 International Conference on Document Analysis and Recognition*. IEEE. 2011, pp. 1135–1139.

[10] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011.

[11] Pham Tanh Dat et al. "Classifier training from a generative model". In: *2019 International Conference on Content-Based Multimedia Indexing (CBMI)*. IEEE. 2019.

[12] Harm De Vries et al. "Modulating early visual processing by language". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6594–6604.

[13] Jia Deng, Alexander C Berg, and Li Fei-Fei. "Hierarchical semantic indexing for large scale image retrieval". In: *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. IEEE. 2011, pp. 785–792.

[14] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics, June 2019, pp. 4171–4186. URL: https://www.aclweb.org/anthology/N19-1423.

[15] Terrance Devries and Graham W. Taylor. "Improved Regularization of Convolutional Neural Networks with Cutout". In: *CoRR* abs/1708.04552 (2017). arXiv: 1708.04552. URL: http://arxiv.org/abs/1708.04552.

[16] Anuvabh Dutt. "Towards Incremental Learning with Deep Convolutional Networks". In: *Conférence en Recherche d'Informations et Applications - CORIA 2017, 14th French Information Retrieval Conference, Marseille, France, March 29-31, 2017. Proceedings*. 2017, pp. 385–394. URL: https://doi.org/10.24348/coria.2017.RJCRI%5C_4.

[17] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled Ensembles of Neural Networks". In: *2018 International Conference on Content-Based Multimedia Indexing (CBMI)*. IEEE. 2018, pp. 1–6.

[18] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled ensembles of neural networks". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=rylRCUJDG.

[19] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Coupled ensembles of neural networks". In: *Neurocomputing* (2019). URL: https://doi.org/10.1016/j.neucom.2018.10.092.

[20] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Improving Hierarchical Image Classification with Merged CNN Architectures". In: *Proceedings of the 15th International Workshop on Content-Based Multimedia Indexing, CBMI 2017, Florence, Italy, June 19-21, 2017*. 2017, 31:1–31:7.

[21] Anuvabh Dutt, Denis Pellerin, and Georges Quénot. "Improving image classification using coarse and fine labels". In: *Proceedings of the 2017 ACM on International Conference on Multimedia Retrieval*. ACM. 2017, pp. 438–442.

[22] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJl-b3RcF7.

[23] Yoav Freund and Robert E Schapire. "A Short Introduction to Boosting". In: *Journal of Japanese Society for Artificial Intelligence* 14.5 (1999), pp. 771–780.

[24] Xavier Gastaldi. "Shake-shake regularization". In: *International Conference on Learning Representations (Workshop)* (2017).

[25] Jonas Gehring et al. "Convolutional sequence to sequence learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1243–1252.

[26] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[27] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. "Qualitatively characterizing neural network optimization problems". In: *International Conference on Learning Representations*. 2015.

[28] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[29] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[30] Klaus Greff, Rupesh K Srivastava, and Jürgen Schmidhuber. "Highway and Residual Networks learn Unrolled Iterative Estimation". In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=Skn9Shcxe.

[31] Chuan Guo et al. "On Calibration of Modern Neural Networks". In: *International Conference on Machine Learning*. 2017, pp. 1321–1330.

[32] Yunhui Guo et al. "Depthwise Convolution is All You Need for Learning Multiple Visual Domains". In: *Thirty-Third AAAI Conference on Artificial Intelligence*. 2019.

[33] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *International Conference on Learning Representations*. 2015.

[34] Lars Kai Hansen and Peter Salamon. "Neural network ensembles". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12.10 (1990), pp. 993–1001.

[35] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[36] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[37] Kaiming He et al. "Identity mappings in deep residual networks". In: *European Conference on Computer Vision*. Springer. 2016, pp. 630–645.

[38] Xu He and Herbert Jaeger. "Overcoming Catastrophic Interference using Conceptor Aided Backpropagation". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=B1al7jg0b.

[39] Martin Heusel et al. "Gans trained by a two time-scale update rule converge to a local nash equilibrium". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6626–6637.

[40] Geoffrey Hinton and David C Plaut. "Using fast weights to deblur old memories". In: *Proceedings of the ninth annual conference of the Cognitive Science Society*. 1987.

[41] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network". In: *NIPS 2014 Deep Learning Workshop* (2014).

[42] Geoffrey Hinton et al. "Deep neural networks for acoustic modeling in speech recognition". In: *IEEE Signal processing magazine* 29 (2012).

[43] Sepp Hochreiter and Jürgen Schmidhuber. "Flat minima". In: *Neural Computation* 9.1 (1997), pp. 1–42.

[44] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[45] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[46] Gao Huang et al. "Snapshot ensembles: Train 1, get m for free". In: *International Conference on Learning Representations*. 2017.

[47] Sergey Ioffe. "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models". In: *Advances in neural information processing systems*. 2017, pp. 1945–1953.

[48] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015, pp. 448–456.

[49] Yangqing Jia and Trevor Darrell. "Latent task adaptation with large-scale hierarchies". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2013, pp. 2080–2087.

[50] Kenji Kawaguchi. "Deep learning without poor local minima". In: *Advances in neural information processing systems*. 2016, pp. 586–594.

[51] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *International Conference on Learning Representations*. 2015.

[52] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *Proceedings of the National Academy of Sciences* (2017).

[53] Alex Krizhevsky and Geoffrey Hinton. "Learning multiple layers of features from tiny images". In: *University of Toronto, Technical report* (2009).

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[55] Anders Krogh and John A Hertz. "A simple weight decay can improve generalization". In: *Advances in neural information processing systems*. 1992, pp. 950–957.

[56] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and scalable predictive uncertainty estimation using deep ensembles". In: *Advances in Neural Information Processing Systems*. 2017, pp. 6402–6413.

[57] Yann LeCun, John S Denker, and Sara A Solla. "Optimal brain damage". In: *Advances in neural information processing systems*. 1990, pp. 598–605.

[58] Yann LeCun et al. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[59] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[60] Hao Li et al. "Pruning filters for efficient convnets". In: *International Conference on Learning Representations*. 2017.

[61] Zhizhong Li and Derek Hoiem. "Learning without forgetting". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2017).

[62] Min Lin, Qiang Chen, and Shuicheng Yan. "Network in network". In: *International Conference on Learning Representations*. 2013.

[63] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. 2014, pp. 740–755.

[64] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "DARTS: Differentiable Architecture Search". In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=S1eYHoC5FX.

[65] David Lopez-Paz et al. "Gradient Episodic Memory for Continuum Learning". In: *NIPS*. 2017.

[66] Ilya Loshchilov and Frank Hutter. "Sgdr: Stochastic gradient descent with warm restarts". In: *International Conference on Learning Representations*. 2016.

[67] Vladimir Macko et al. "Improving neural architecture search image classifiers via ensemble learning". In: *arXiv preprint arXiv:1903.06236* (2019).

[68] Arun Mallya and Svetlana Lazebnik. "Piggyback: Adding Multiple Tasks to a Single, Fixed Network by Learning to Mask". In: *ECCV* (2018).

[69] Michael McCloskey and Neal J Cohen. "Catastrophic interference in connectionist networks: The sequential learning problem". In: *Psychology of learning and motivation* 24 (1989).

[70] Mehdi Mirza and Simon Osindero. "Conditional generative adversarial nets". In: *arXiv preprint arXiv:1411.1784* (2014).

[71] Dmytro Mishkin and Jiri Matas. "All you need is a good init". In: *International Conference on Learning Representations*. 2016.

[72] Takeru Miyato et al. "Spectral Normalization for Generative Adversarial Networks". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=B1QRgziT-.

[73] Nelson Morgan and Hervé Bourlard. "Generalization and parameter estimation in feedforward nets: Some experiments". In: *Advances in neural information processing systems*. 1990, pp. 630–637.

[74] Yuval Netzer et al. "Reading digits in natural images with unsupervised feature learning". In: *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011).

[75] Tianyu Pang et al. "Improving Adversarial Robustness via Promoting Ensemble Diversity". In: *Proceedings of the 36th International Conference on Machine Learning*. Vol. 97. Proceedings of Machine Learning Research. 2019, pp. 4970–4979. URL: http://proceedings.mlr.press/v97/pang19a.html.

[76] Nicolas Papernot et al. "Semi-supervised Knowledge Transfer for Deep Learning from Private Training Data". In: *International Conference on Learning Representations*. 2016.

[77] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *International Conference on Learning Representations*. 2016.

[78] Roger Ratcliff. "Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions". In: *Psychological review* 97.2 (1990).

[79] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. "Learning multiple visual domains with residual adapters". In: *NIPS*. 2017.

[80] Sylvestre-Alvise Rebuffi et al. "icarl: Incremental classifier and representation learning". In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2017, pp. 2001–2010.

[81] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[82] Mark B Ring. "CHILD: A first step towards continual learning". In: *Machine Learning* 28.1 (1997), pp. 77–104.

[83] Anthony Robins. "Catastrophic forgetting, rehearsal and pseudorehearsal". In: *Connection Science* 7.2 (1995), pp. 123–146.

[84] Amir Rosenfeld and John K Tsotsos. "Incremental learning through deep adaptation". In: *IEEE transactions on pattern analysis and machine intelligence* (2018).

[85] Amelie Royer and Christoph H Lampert. "Classifier adaptation at prediction time". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 1401–1409.

[86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning internal representations by error propagation". In: *Parallel Distributed Processing*. 1985.

[87] Olga Russakovsky et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[88] Andrei A Rusu et al. "Progressive neural networks". In: *CoRR* (2016). arXiv: 1606.04671. URL: http://arxiv.org/abs/1606.04671.

[89] Tim Salimans and Durk P Kingma. "Weight normalization: A simple reparameterization to accelerate training of deep neural networks". In: *Advances in Neural Information Processing Systems*. 2016, pp. 901–909.

[90] Tim Salimans et al. "Improved techniques for training gans". In: *Advances in neural information processing systems*. 2016, pp. 2234–2242.

[91] Shibani Santurkar, Ludwig Schmidt, and Aleksander Madry. "A Classification Based Study of Covariate Shift in GAN Distributions". In: *International Conference on Machine Learning*. 2018, pp. 4487–4496.

[92] Shibani Santurkar et al. "How does batch normalization help optimization?" In: *Advances in Neural Information Processing Systems*. 2018, pp. 2483–2493.

[93] Hanul Shin et al. "Continual learning with deep generative replay". In: *Advances in Neural Information Processing Systems*. 2017, pp. 2990–2999.

[94] Konstantin Shmelkov, Cordelia Schmid, and Karteek Alahari. "How good is my GAN?" In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 213–229.

[95] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[96] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *International Conference on Learning Representations*. 2015.

[97] Jost Tobias Springenberg et al. "Striving for simplicity: The all convolutional net". In: *International Conference on Learning Representations (Workshop)* (2015).

[98] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[99] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Training very deep networks". In: *Advances in neural information processing systems*. 2015, pp. 2377–2385.

[100] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[101] Mingxing Tan et al. "Mnasnet: Platform-aware neural architecture search for mobile". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[102] Sebastian Thrun and Tom M Mitchell. "Lifelong robot learning". In: *The biology and technology of intelligent autonomous agents*. Springer, 1995.

[103] Stefen Chan Wai Tim et al. "Descriptor extraction based on a multilayer dictionary architecture for classification of natural images". In: *Computer Vision and Image Understanding* (2018). URL: https://doi.org/10.1016/j.cviu.2018.08.002.

[104] Andreas Veit, Michael J Wilber, and Serge Belongie. "Residual networks behave like ensembles of relatively shallow networks". In: *Advances in neural information processing systems*. 2016, pp. 550–558.

[105] Xiaolong Wang et al. "Non-local neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7794–7803.

[106] Yuxin Wu and Kaiming He. "Group normalization". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 3–19.

[107] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500.

[108] Zhicheng Yan et al. "HD-CNN: hierarchical deep convolutional neural networks for large scale visual recognition". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2740–2748.

[109] Jaehong Yoon et al. "Lifelong Learning with Dynamically Expandable Networks". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=Sk7KsfW0-.

[110] Han Zhang et al. "Self-Attention Generative Adversarial Networks". In: *International Conference on Machine Learning*. 2019, pp. 7354–7363.

[111] Ting Zhang et al. "Interleaved group convolutions". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 4373–4382.

[112] Liming Zhao et al. "On the Connection of Deep Fusion to Ensembling". In: *CoRR* abs/1611.07718 (2016). arXiv: 1611.07718. URL: http://arxiv.org/abs/1611.07718.

[113] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *International Conference on Learning Representations*. 2017.

[114] Barret Zoph et al. “Learning transferable architectures for scalable image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 8697–8710.